Contents

errno Constants

Exception-Handling Constants

EXIT_SUCCESS, EXIT_FAILURE

File Attribute Constants

File Constants

File Permission Constants

File Read-Write Access Constants

File Translation Constants

FILENAME_MAX

FOPEN_MAX, _SYS_OPEN

_FREEENTRY, _USEDENTRY

fseek, _lseek Constants

Heap Constants

_HEAP_MAXREQ

HUGE_VAL, _HUGE

Locale Categories

_locking Constants

Math Constants

Math Error Constants

_MAX_ENV

MB_CUR_MAX

NULL (CRT)

Path Field Limits

RAND_MAX

setvbuf Constants

Sharing Constants

signal Constants

signal Action Constants

spawn Constants

_stat Structure st_mode Field Constants

stdin, stdout, stderr

TMP_MAX, L_tmpnam

_aligned_offset_realloc_dbg

_aligned_offset_recalloc

_aligned_offset_recalloc_dbg

_aligned_realloc

_aligned_realloc_dbg

_aligned_recalloc

_aligned_recalloc_dbg

_alloca

_amsg_exit

and

and_eq

asctime, _wasctime

asctime_s, _wasctime_s

asin, asinf, asinl

asinh, asinhf, asinhl

assert Macro, _assert, _wassert

_ASSERT, _ASSERTE, _ASSERT_EXPR Macros

atan, atanf, atanl, atan2, atan2f, atan2l

atanh, atanhf, atanhl

atexit

_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

atof, _atof_l, _wtof, _wtof_l

atoi, _atoi_l, _wtoi, _wtoi_l

_atoi64, _atoi64_l, _wtoi64, _wtoi64_l

atol, _atol_l, _wtol, _wtol_l

atoll, _atoll_l, _wtoll, _wtoll_l

_beginthread, _beginthreadex

Bessel Functions: _j0, _j1, _jn, _y0, _y1, _yn

bitand

bitor

bsearch

bsearch_s

btowc

_byteswap_uint64, _byteswap_ulong, _byteswap_ushort

c16rtomb, c32rtomb

cabs, cabsf, cabsl

_cabs

cacos, cacosf, cacosl

cacosh, cacoshf, cacoshl

_callnewh

calloc

_calloc_dbg

carg, cargf, cargl

casin, casinf, casinl

casinh, casinhf, casinhl

catan, catanf, catanl

catanh, catanhf, catanhl

cbrt, cbrtf, cbrtl

_Cbuild, _FCbuild, _LCbuild

ccos, ccosf, ccosl

ccosh, ccoshf, ccoshl

ceil, ceilf, ceill

_cexit, _c_exit

cexp, cexpf, cexpl

cgets

_cgets_s, _cgetws_s

chdir

_chdir, _wchdir

_chdrive

_chgsign, _chgsignf, _chgsignl

chmod

_chmod, _wchmod

chsize

_chsize

_creat, _wcreat

_create_locale, _wcreate_locale

_CrtCheckMemory

_CrtDbgBreak

_CrtDbgReport, _CrtDbgReportW

_CrtDoForAllClientObjects

_CrtDumpMemoryLeaks

_CrtGetAllocHook

_CrtGetDumpClient

_CrtGetReportHook

_CrtIsMemoryBlock

_CrtIsValidHeapPointer

_CrtIsValidPointer

_CrtMemCheckpoint

_CrtMemDifference

_CrtMemDumpAllObjectsSince

_CrtMemDumpStatistics

_CrtReportBlockType

_CrtSetAllocHook

_CrtSetBreakAlloc

_CrtSetDbgFlag

_CrtSetDebugFillThreshold

_CrtSetDumpClient

_CrtSetReportFile

_CrtSetReportHook

_CrtSetReportHook2, _CrtSetReportHookW2

_CrtSetReportMode

cscanf

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l

csin, csinf, csinl

csinh, csinhf, csinhl

csqrt, csqrtf, csqrtl

ctan, ctanf, ctanl

ctanh, ctanhf, ctanhl

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64

ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s

cwait

_cwait

_CxxThrowException

difftime, _difftime32, _difftime64

div

dup, dup2

_dup, _dup2

_dupenv_s, _wdupenv_s

_dupenv_s_dbg, _wdupenv_s_dbg

ecvt

_ecvt

_ecvt_s

_endthread, _endthreadex

eof

_eof

erf, erff, erfl, erfc, erfcf, erfcl

execl

_execl, _wexecl

execle

_execle, _wexecle

execlp

_execlp, _wexeclp

execlpe

_execlpe, _wexeclpe

execv

_execv, _wexecv

execve

fprintf, _fprintf_l, fwprintf, _fwprintf_l

_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l

fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l

fputc, fputwc

_fputc_nolock, _fputwc_nolock

fputchar

_fputchar, _fputwchar

fputs, fputws

fread

fread_s

_fread_nolock

_fread_nolock_s2

free

_free_dbg

_free_locale

_freea

freopen, _wfreopen

freopen_s, _wfreopen_s

frexp

fscanf, _fscanf_l, fwscanf, _fwscanf_l

fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fseek, _fseeki64

_fseek_nolock, _fseeki64_nolock

fsetpos

_fsopen, _wfsopen

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

ftell, _ftelli64

_ftell_nolock, _ftelli64_nolock

_ftime, _ftime32, _ftime64

_ftime_s, _ftime32_s, _ftime64_s

_fullpath, _wfullpath

_fullpath_dbg, _wfullpath_dbg

_futime, _futime32, _futime64

fwide

fwrite

_fwrite_nolock

gcvt

_gcvt

_gcvt_s

_get_current_locale

_get_daylight

_get_doserrno

_get_dstbias

_get_errno

_get_FMA3_enable, _set_FMA3_enable

_get_fmode

_get_heap_handle

_get_invalid_parameter_handler, _get_thread_local_invalid_parameter_handler

_get_osfhandle

_get_pgmptr

_get_printf_count_output

_get_purecall_handler, _set_purecall_handler

_get_terminate

_get_timezone

_get_tzname

_get_unexpected

_get_wpgmptr

getc, getwc

_getc_nolock, _getwc_nolock

getch

_getch, _getwch

_getch_nolock, _getwch_nolock

getchar, getwchar

_getchar_nolock, _getwchar_nolock

getche

_getche, _getwche

_getche_nolock, _getwche_nolock

getcwd

_getcwd, _wgetcwd

_getcwd_dbg, _wgetcwd_dbg

_getdcwd, _wgetdcwd

_getdcwd_dbg, _wgetdcwd_dbg

_getdcwd_nolock, _wgetdcwd_nolock

_getdiskfree

_getdrive

_getdrives

getenv, _wgetenv

getenv_s, _wgetenv_s

_getmaxstdio

_getmbcp

getpid

_getpid

gets_s, _getws_s

getw

_getw

gmtime, _gmtime32, _gmtime64

gmtime_s, _gmtime32_s, _gmtime64_s

_heapchk

_heapmin

_heapwalk

hypot, hypotf, hypotl, _hypot, _hypotf, _hypotl

ilogb, ilogbf, ilogbl2

imaxabs

imaxdiv

_initterm, _initterm_e

_invalid_parameter, _invalid_parameter_noinfo, _invalid_parameter_noinfo_noreturn, _invoke_watson

isalnum, iswalnum, _isalnum_l, _iswalnum_l

isalpha, iswalpha, _isalpha_l, _iswalpha_l

isascii, __isascii, iswascii

isatty

_isatty

isblank, iswblank, _isblank_l, _iswblank_l

iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l

_isctype, iswctype, _isctype_l, _iswctype_l

iscsym, iscsymf, __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l

isdigit, iswdigit, _isdigit_l, _iswdigit_l

isfinite, _finite, _finitef

isgraph, iswgraph, _isgraph_l, _iswgraph_l

isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered

isinf

isleadbyte, _isleadbyte_l

islower, iswlower, _islower_l, _iswlower_l

_ismbbalnum, _ismbbalnum_l

_ismbbalpha, _ismbbalpha_l

_ismbbblank, _ismbbblank_l

_ismbbgraph, _ismbbgraph_l

_ismbbkalnum, _ismbbkalnum_l

_ismbbkana, _ismbbkana_l

_ismbbkprint, _ismbbkprint_l

_ismbbkpunct, _ismbbkpunct_l

_ismbblead, _ismbblead_l

_ismbbprint, _ismbbprint_l

_ismbbpunct, _ismbbpunct_l

_ismbbtrail, _ismbbtrail_l

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l

_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l

_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l

_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l

_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

_ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l

isnan, _isnan, _isnanf

isnormal

ispunct, iswpunct, _ispunct_l, _iswpunct_l

isprint, iswprint, _isprint_l, _iswprint_l

isspace, iswspace, _isspace_l, _iswspace_l

isupper, _isupper_l, iswupper, _iswupper_l

isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l

itoa, _itoa, ltoa, _ltoa, ultoa, _ultoa, _i64toa, _ui64toa, _itow, _ltow, _ultow, _i64tow, _ui64tow

_itoa_s, _ltoa_s, _ultoa_s, _i64toa_s, _ui64toa_s, _itow_s, _ltow_s, _ultow_s, _i64tow_s, _ui64tow_s

j0, j1, jn

kbhit

_kbhit

ldexp

ldiv, lldiv

lfind

_lfind

_lfind_s

lgamma, lgammaf, lgammal

localeconv

localtime, _localtime32, _localtime64

localtime_s, _localtime32_s, _localtime64_s

_lock_file

locking

_locking

log, logf, log10, log10f

log1p, log1pf, log1pl2

log2, log2f, log2l

logb, logbf, logbl, _logb, _logbf

longjmp

lrint, lrintf, lrintl, llrint, llrintf, llrintl

lround, lroundf, lroundl, llround, llroundf, llroundl

_lrotl, _lrotr

lsearch

_lsearch

_lsearch_s

lseek

_lseek, _lseeki64

_makepath, _wmakepath

_makepath_s, _wmakepath_s

malloc

_malloc_dbg

_malloca

_matherr

__max

_mbbtombc, _mbbtombc_l

_mbbtype, _mbbtype_l

_mbccpy, _mbccpy_l

_mbccpy_s, _mbccpy_s_l

_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l

_mbclen, mblen, _mblen_l, _mbclen_l

_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l

_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

_mbctombb, _mbctombb_l

mbrlen

mbrtoc16, mbrtoc323

mbrtowc

_mbsbtype, _mbsbtype_l

mbsinit

_mbsnbcat, _mbsnbcat_l

_mbsnbcat_s, _mbsnbcat_s_l

_mbsnbcmp, _mbsnbcmp_l

_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l

_mbsnbcpy, _mbsnbcpy_l

_mbsnbcpy_s, _mbsnbcpy_s_l

_mbsnbicmp, _mbsnbicmp_l

_mbsnbset, _mbsnbset_l

_mbsnbset_s, _mbsnbset_s_l

mbsrtowcs

mbsrtowcs_s

mbstowcs, _mbstowcs_l

mbstowcs_s, _mbstowcs_s_l

mbtowc, _mbtowc_l

memccpy

_memccpy

memchr, wmemchr

memcmp, wmemcmp

memcpy, wmemcpy

memcpy_s, wmemcpy_s

memicmp

_memicmp, _memicmp_l

memmove, wmemmove

memmove_s, wmemmove_s

memset, wmemset

__min

mkdir

_mkdir, _wmkdir

_mkgmtime, _mkgmtime32, _mkgmtime64

mktemp

_mktemp, _wmktemp

_mktemp_s, _wmktemp_s

mktime, _mktime32, _mktime64

modf, modff, modfl

_msize

_msize_dbg

nan, nanf, nanl

nearbyint, nearbyintf, nearbyintl

nextafter, nextafterf, nextafterl, _nextafter, _nextafterf, nexttoward, nexttowardf, nexttowardl

norm, normf, norml

not

not_eq

offsetof Macro

_onexit, _onexit_m

open

_open, _wopen

_open_osfhandle

or_eq

or

_pclose

perror, _wperror

_pipe

_popen, _wpopen

pow, powf, powl

printf, _printf_l, wprintf, _wprintf_l

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

printf_s, _printf_s_l, wprintf_s, _wprintf_s_l

_purecall

putc, putwc

_putc_nolock, _putwc_nolock

putch

_putch, _putwch

_putch_nolock, _putwch_nolock

putchar, putwchar

_putchar_nolock, _putwchar_nolock

putenv

_putenv, _wputenv

_putenv_s, _wputenv_s

puts, _putws

putw

_putw

_query_new_handler

_query_new_mode

quick_exit

qsort

qsort_s

raise

rand

rand_s

read

_read

realloc

_realloc_dbg

_recalloc

_recalloc_dbg

remainder, remainderf, remainderl

remove, _wremove

remquo, remquof, remquol

rename, _wrename

_resetstkoflw

rewind

rint, rintf, rintl

rmdir

_rmdir, _wrmdir

rmtmp

_rmtmp

_rotl, _rotl64, _rotr, _rotr64

round, roundf, roundl

_RPT, _RPTF, _RPTW, _RPTFW Macros

_RTC_GetErrDesc

_RTC_NumErrors

_RTC_SetErrorFunc

_RTC_SetErrorFuncW

_RTC_SetErrorType

_scalb

scalbn, scalbnf, scalbnl, scalbln, scalblnf, scalblnl

scanf, _scanf_l, wscanf, _wscanf_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

_scprintf, _scprintf_l, _scwprintf, _scwprintf_l

_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l

_searchenv, _wsearchenv

_searchenv_s, _wsearchenv_s

__security_init_cookie

_seh_filter_dll, _seh_filter_exe

_set_abort_behavior

setbuf

_set_controlfp

_set_doserrno

_set_errno

_set_error_mode

_set_fmode

_set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler

setjmp

setlocale, _wsetlocale

_setmaxstdio

_setmbcp

setmode

_setmode

_set_new_handler

spawnvpe

_spawnvpe, _wspawnvpe

_splitpath, _wsplitpath

_splitpath_s, _wsplitpath_s

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l

sqrt, sqrtf, sqrtl

srand

sscanf, _sscanf_l, swscanf, _swscanf_l

sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

_stat, _stat32, _stat64, _stati64, _stat32i64, _stat64i32, _wstat, _wstat32, _wstat64, _wstati64, _wstat32i64, _wstat64i32

_STATIC_ASSERT Macro

_status87, _statusfp, _statusfp2

strcat, wcscat, _mbscat

strcat_s, wcscat_s, _mbscat_s, _mbscat_s_l

strchr, wcschr, _mbschr, _mbschr_l

strcmp, wcscmp, _mbscmp, _mbscmp_l

strcmpi

strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l

strcpy, wcscpy, _mbscpy

strcpy_s, wcscpy_s, _mbscpy_s, _mbscpy_s_l

strcspn, wcscspn, _mbscspn, _mbscspn_l

_strdate, _wstrdate

_strdate_s, _wstrdate_s

_strdec, _wcsdec, _mbsdec, _mbsdec_l

strdup, wcsdup

_strdup, _wcsdup, _mbsdup

_strdup_dbg, _wcsdup_dbg

strerror, _strerror, _wcserror, __wcserror

strerror_s, _strerror_s, _wcserror_s, __wcserror_s

strftime, wcsftime, _strftime_l, _wcsftime_l

stricmp, wcsicmp

_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l

_strinc, _wcsinc, _mbsinc, _mbsinc_l

strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

strlwr, wcslwr

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l

_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l

strnicmp, wcsnicmp

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l

_strninc, _wcsninc, _mbsninc, _mbsninc_l

strnlen, strnlen_s, wcsnlen, wcsnlen_s, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

strnset, wcsnset

_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

_strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strrev, wcsrev

_strrev, _wcsrev, _mbsrev, _mbsrev_l

strset, wcsset

_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l

strspn, wcsspn, _mbsspn, _mbsspn_l

_strspnp, _wcsspnp, _mbsspnp, _mbsspnp_l

strstr, wcsstr, _mbsstr, _mbsstr_l

_strtime, _wstrtime

_strtime_s, _wstrtime_s

strtod, _strtod_l, wcstod, _wcstod_l

strtof, _strtof_l, wcstof, _wcstof_l

_strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l

strtoimax, _strtoimax_l, wcstoimax, _wcstoimax_l

strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l

strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l

strtol, wcstol, _strtol_l, _wcstol_l

strtold, _strtold_l, wcstold, _wcstold_l

strtoll, _strtoll_l, wcstoll, _wcstoll_l

_strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

strtoul, _strtoul_l, wcstoul, _wcstoul_l

strtoull, _strtoull_l, wcstoull, _wcstoull_l

strtoumax, _strtoumax_l, wcstoumax, _wcstoumax_l

strupr, wcsupr

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

swab

_swab

system, _wsystem

tan, tanf, tanl

tanh, tanhf, tanhl

tell

_tell, _telli64

tempnam

_tempnam, _wtempnam, tmpnam, _wtmpnam

_tempnam_dbg, _wtempnam_dbg

terminate (CRT)

tgamma, tgammaf, tgammal

time, _time32, _time64

timespec_get, _timespec32_get, _timespec64_get

tmpfile

tmpfile_s

tmpnam_s, _wtmpnam_s

toascii, __toascii

tolower, _tolower, towlower, _tolower_l, _towlower_l

toupper, _toupper, towupper, _toupper_l, _towupper_l

towctrans

trunc, truncf, truncl

tzset

_tzset

umask

_umask

_umask_s

__uncaught_exception

unexpected (CRT)

ungetc, ungetwc

_ungetc_nolock, _ungetwc_nolock

ungetch

_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

unlink

_unlink, _wunlink

_unlock_file

_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64

va_arg, va_copy, va_end, va_start

_vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l

_vcprintf_p, _vcprintf_p_l, _vcwprintf_p, _vcwprintf_p_l

_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l

vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l

vfscanf, vfwscanf

vfscanf_s, vfwscanf_s

vprintf, _vprintf_l, vwprintf, _vwprintf_l

_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l

vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l

vscanf, vwscanf

vscanf_s, vwscanf_s

_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

_vscprintf_p, _vscprintf_p_l, _vscwprintf_p, _vscwprintf_p_l

vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l

_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l

vsscanf, vswscanf

vsscanf_s, vswscanf_s

wcrtomb

wcrtomb_s

wcsrtombs

wcsrtombs_s

wcstombs, _wcstombs_l

wcstombs_s, _wcstombs_s_l

wctob

wctomb, _wctomb_l

wctomb_s, _wctomb_s_l

wctrans

wctype

write

_write

wcsicoll

xor

xor_eq

y0, y1, yn

C Run-Time Library Reference

10/31/2018 • 2 minutes to read • Edit Online

The Microsoft run-time library provides routines for programming for the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

Sample programs are included in the individual reference topics for most routines in the library.

In This Section

C Run-Time Libraries
Discusses the .lib files that comprise the C run-time libraries.

Universal C runtime routines by category
Provides links to the run-time library by category.

Global Variables and Standard Types
Provides links to the global variables and standard types provided by the run-time library.

Global Constants
Provides links to the global constants defined by the run-time library.

Alphabetical Function Reference
Provides a table of contents entry point into an alphabetical listing of all C run-time library functions.

Generic-Text Mappings
Provides links to the generic-text mappings defined in Tchar.h.

Language and Country/Region Strings
Describes how to use the `setlocale` function to set the language and Country/Region strings.

Related Sections

Debug Routines
Provides links to the debug versions of the run-time library routines.

Run-Time Error Checking
Provides links to functions that support run-time error checks.

DLLs and Visual C++ run-time library behavior
Discusses the entry point and startup code used for a DLL.

Debugging
Provides links to using the Visual Studio debugger to correct logic errors in your application or stored procedures.

CRT Library Features

4/1/2019 • 8 minutes to read • Edit Online

This topic discusses the various .lib files that comprise the C run-time libraries as well as their associated compiler options and preprocessor directives.

C Run-Time Libraries (CRT)

The C Run-time Library (CRT) is the part of the C++ Standard Library that incorporates the ISO C99 standard library. The Visual C++ libraries that implement the CRT support native code development, and both mixed native and managed code. All versions of the CRT support multi-threaded development. Most of the libraries support both static linking, to link the library directly into your code, or dynamic linking to let your code use common DLL files.

Starting in Visual Studio 2015, the CRT has been refactored into new binaries. The Universal CRT (UCRT) contains the functions and globals exported by the standard C99 CRT library. The UCRT is now a Windows component, and ships as part of Windows 10. The static library, DLL import library, and header files for the UCRT are now found in the Windows 10 SDK. When you install Visual C++, Visual Studio setup installs the subset of the Windows 10 SDK required to use the UCRT. You can use the UCRT on any version of Windows supported by Visual Studio 2015 and later versions. You can redistribute it using vcredist for supported versions of Windows other than Windows 10. For more information, see Redistributing Visual C++ Files.

The following table lists the libraries that implement the UCRT.

LIBRARY	ASSOCIATED DLL	CHARACTERISTICS	OPTION	PREPROCESSOR DIRECTIVES
libucrt.lib	None	Statically links the UCRT into your code.	**/MT**	_MT
libucrtd.lib	None	Debug version of the UCRT for static linking. Not redistributable.	**/MTd**	_DEBUG, _MT
ucrt.lib	ucrtbase.dll	DLL import library for the UCRT.	**/MD**	_MT, _DLL
ucrtd.lib	ucrtbased.dll	DLL import library for the Debug version of the UCRT. Not redistributable.	**/MDd**	_DEBUG, _MT, _DLL

The vcruntime library contains Visual C++ CRT implementation-specific code, such as exception handling and debugging support, runtime checks and type information, implementation details and certain extended library functions. This library is specific to the version of the compiler used.

This table lists the libraries that implement the vcruntime library.

LIBRARY	ASSOCIATED DLL	CHARACTERISTICS	OPTION	PREPROCESSOR DIRECTIVES
libvcruntime.lib	None	Statically linked into your code.	**/MT**	_MT
libvcruntimed.lib	None	Debug version for static linking. Not redistributable.	**/MTd**	_MT, _DEBUG
vcruntime.lib	vcruntime<version>.dll	DLL import library for the vcruntime.	**/MD**	_MT, _DLL
vcruntimed.lib	vcruntime<version>d.dll	DLL import library for the Debug vcruntime. Not redistributable.	**/MDd**	_DEBUG, _MT, _DLL

NOTE

When the UCRT refactoring occurred, the Concurrency Runtime functions were moved into concrt140.dll, which was added to the C++ redistributable package. This DLL is required for C++ parallel containers and algorithms such as `concurrency::parallel_for`. In addition, the C++ Standard Library requires this DLL on Windows XP to support synchronization primitives, because Windows XP does not have condition variables.

The code that initializes the CRT is in one of several libraries, based on whether the CRT library is statically or dynamically linked, or native, managed, or mixed code. This code handles CRT startup, internal per-thread data initialization, and termination. It is specific to the version of the compiler used. This library is always statically linked, even when using a dynamically linked UCRT.

This table lists the libraries that implement CRT initialization and termination.

LIBRARY	CHARACTERISTICS	OPTION	PREPROCESSOR DIRECTIVES
libcmt.lib	Statically links the native CRT startup into your code.	**/MT**	_MT
libcmtd.lib	Statically links the Debug version of the native CRT startup. Not redistributable.	**/MTd**	_DEBUG, _MT
msvcrt.lib	Static library for the native CRT startup for use with DLL UCRT and vcruntime.	**/MD**	_MT, _DLL
msvcrtd.lib	Static library for the Debug version of the native CRT startup for use with DLL UCRT and vcruntime. Not redistributable.	**/MDd**	_DEBUG, _MT, _DLL

LIBRARY	CHARACTERISTICS	OPTION	PREPROCESSOR DIRECTIVES
msvcmrt.lib	Static library for the mixed native and managed CRT startup for use with DLL UCRT and vcruntime.	**/clr**	
msvcmrtd.lib	Static library for the Debug version of the mixed native and managed CRT startup for use with DLL UCRT and vcruntime. Not redistributable.	**/clr**	
msvcurt.lib	**Deprecated** Static library for the pure managed CRT.	**/clr:pure**	
msvcurtd.lib	**Deprecated** Static library for the Debug version of the pure managed CRT. Not redistributable.	**/clr:pure**	

If you link your program from the command line without a compiler option that specifies a C run-time library, the linker will use the statically linked CRT libraries: libcmt.lib, libvcruntime.lib, and libucrt.lib.

Using the statically linked CRT implies that any state information saved by the C runtime library will be local to that instance of the CRT. For example, if you use strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l when using a statically linked CRT, the position of the `strtok` parser is unrelated to the `strtok` state used in code in the same process (but in a different DLL or EXE) that is linked to another instance of the static CRT. In contrast, the dynamically linked CRT shares state for all code within a process that is dynamically linked to the CRT. This concern does not apply if you use the new more secure versions of these functions; for example, `strtok_s` does not have this problem.

Because a DLL built by linking to a static CRT will have its own CRT state, it is not recommended to link statically to the CRT in a DLL unless the consequences of this are specifically desired and understood. For example, if you call _set_se_translator in an executable that loads the DLL linked to its own static CRT, any hardware exceptions generated by the code in the DLL will not be caught by the translator, but hardware exceptions generated by code in the main executable will be caught.

If you are using the **/clr** compiler switch, your code will be linked with a static library, msvcmrt.lib. The static library provides a proxy between your managed code and the native CRT. You cannot use the statically linked CRT (**/MT** or **/MTd** options) with **/clr**. Use the dynamically-linked libraries (**/MD** or **/MDd**) instead. The pure managed CRT libraries are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017.

For more information on using the CRT with **/clr**, see Mixed (Native and Managed) Assemblies.

To build a debug version of your application, the _DEBUG flag must be defined and the application must be linked with a debug version of one of these libraries. For more information about using the debug versions of the library files, see CRT Debugging Techniques.

This version of the CRT is not fully conformant with the C99 standard. In particular, the <tgmath.h> header and the CX_LIMITED_RANGE/FP_CONTRACT pragma macros are not supported. Certain elements such as the meaning of parameter specifiers in standard IO functions use legacy interpretations by default. You can use /Zc compiler conformance options and specify linker options to control some aspects of library conformance,

C++ Standard Library

C++ STANDARD LIBRARY	CHARACTERISTICS	OPTION	PREPROCESSOR DIRECTIVES
libcpmt.lib	Multithreaded, static link	**/MT**	_MT
msvcprt.lib	Multithreaded, dynamic link (import library for MSVCP*version*.dll)	**/MD**	_MT, _DLL
libcpmtd.lib	Multithreaded, static link	**/MTd**	_DEBUG, _MT
msvcprtd.lib	Multithreaded, dynamic link (import library for MSVCP*version*D.DLL)	**/MDd**	_DEBUG, _MT, _DLL

When you build a release version of your project, one of the basic C run-time libraries (libcmt.lib, msvcmrt.lib, msvcrt.lib) is linked by default, depending on the compiler option you choose (multithreaded, DLL, /clr). If you include one of the C++ Standard Library header files in your code, a C++ Standard Library will be linked in automatically by Visual C++ at compile time. For example:

```
#include <ios>
```

For binary compatibility, more than one DLL file may be specified by a single import library. Version updates may introduce *dot libraries*, separate DLLs that introduce new library functionality. For example, Visual Studio 2017 version 15.6 introduced msvcp140_1.dll to support additional standard library functionality without breaking the ABI supported by msvcp140.dll. The msvcprt.lib import library included in the toolset for Visual Studio 2017 version 15.6 supports both DLLs, and the vcredist for this version installs both DLLs. Once shipped, a dot library has a fixed ABI, and will never have a dependency on a later dot library.

What problems exist if an application uses more than one CRT version?

Every executable image (EXE or DLL) can have its own statically linked CRT, or can dynamically link to a CRT. The version of the CRT statically included in or dynamically loaded by a particular image depends on the version of the tools and libraries it was built with. A single process may load multiple EXE and DLL images, each with its own CRT. Each of those CRTs may use a different allocator, may have different internal structure layouts, and may use different storage arrangements. This means that allocated memory, CRT resources, or classes passed across a DLL boundary can cause problems in memory management, internal static usage, or layout interpretation. For example, if a class is allocated in one DLL but passed to and deleted by another, which CRT deallocator is used? The errors caused can range from the subtle to the immediately fatal, and therefore direct transfer of such resources is strongly discouraged.

You can avoid many of these issues by using Application Binary Interface (ABI) technologies instead, as they are designed to be stable and versionable. Design your DLL export interfaces to pass information by value, or to work on memory that is passed in by the caller rather than allocated locally and returned to the caller. Use marshalling techniques to copy structured data between executable images. Encapsulate resources locally and only allow manipulation through handles or functions you expose to clients.

It's also possible to avoid some of these issues if all of the images in your process use the same dynamically loaded version of the CRT. To ensure that all components use the same DLL version of the CRT, build them by using the **/MD** option, and use the same compiler toolset and property settings.

Some care is needed if your program passes certain CRT resources (such as file handles, locales and environment variables) across DLL boundaries, even when using the same version of the CRT. For more information on the issues involved and how to resolve them, see Potential Errors Passing CRT Objects Across DLL Boundaries.

See also

- C Run-Time Library Reference

Compatibility

5/8/2019 • 2 minutes to read
• Edit Online

The Universal C Run-Time Library (UCRT) supports most of the C standard library required for C++ conformance. It implements the C99 (ISO/IEC 9899:1999) library, with the exceptions of the type-generic macros defined in <tgmath.h>, and strict type compatibility in <complex.h>. The UCRT also implements a large subset of the POSIX.1 (ISO/IEC 9945-1:1996, the POSIX System Application Program Interface) C library, but is not fully conformant to any specific POSIX standard. In addition, the UCRT implements several Microsoft-specific functions and macros that are not part of a standard.

Functions specific to the Microsoft implementation of Visual C++ are found in the vcruntime library. Many of these functions are for internal use and cannot be called by user code. Some are documented for use in debugging and implementation compatibility.

The C++ standard reserves names that begin with an underscore in the global namespace to the implementation. Because the POSIX functions are in the global namespace, but are not part of the standard C runtime library, the Microsoft-specific implementations of these functions have a leading underscore. For portability, the UCRT also supports the default names, but the Microsoft C++ compiler issues a deprecation warning when code that uses them is compiled. Only the default POSIX names are deprecated, not the functions. To suppress the

warning, define `_CRT_NONSTDC_NO_WARNINGS` before including any headers in code that uses the original POSIX names.

Certain functions in the standard C library have a history of unsafe usage, because of misused parameters and unchecked buffers. These functions are often the source of security issues in code. Microsoft created a set of safer versions of these functions that verify parameter usage and invoke the invalid parameter handler when an issue is detected at runtime. By default, the Microsoft C++ compiler issues a deprecation warning when a function is used that has a safer variant available. When you compile your code as C++ , you can define `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` as 1 to eliminate most warnings. This uses template overloads to call the safer variants while maintaining portable source code. To suppress the warning, define `_CRT_SECURE_NO_WARNINGS` before including any headers in code that uses these functions. For more information, see Security Features in the CRT.

Except as noted within the documentation for specific functions, the UCRT is compatible with the Windows API. Certain functions are not supported in Windows 8 Store apps or in Universal Windows Platform (UWP) apps on Windows 10. These functions are listed in CRT functions not supported in Universal Windows Platform apps, which enumerates the functions not supported by the Windows Runtime and UWP.

Related Articles

TITLE	DESCRIPTION
UWP Apps, the Windows Runtime, and the C Run-Time	Describes when UCRT routines are not compatible with Universal Windows apps or Microsoft Store apps.
ANSI C Compliance	Describes standard-compliant naming in the UCRT.
UNIX	Provides guidelines for porting programs to UNIX.
Windows Platforms (CRT)	Lists the operating systems that are the CRT supports.
Backward Compatibility	Describes how to map old CRT names to the new ones.
CRT Library Features	Provides an overview of the CRT library (.lib) files and the associated compiler options.

UWP Apps, the Windows Runtime, and the C Run-Time

3/11/2019 • 2 minutes to read • Edit Online

Universal Windows Platform (UWP) apps are programs that run in the Windows Runtime that executes on Windows 8. The Windows Runtime is a trustworthy environment that controls the functions, variables, and resources that are available to a UWP app. However, by design, Windows Runtime restrictions prevent the use of most C Run-Time Library (CRT) features in UWP apps.

The Windows Runtime does not support the following CRT features:

- Most CRT functions that are related to unsupported functionality.

  For example, a UWP app cannot create a process by using the **exec** and **spawn** families of routines.

  When a CRT function is not supported in a UWP app, that fact is noted in its reference article.

- Most multibyte character and string functions.

  However, both Unicode and ANSI text are supported.

- Environment variables.

- The concept of a current working directory.

- UWP apps and DLLs that are statically linked to the CRT and built by using the /MT or `/MTd` compiler options.

  That is, an app that uses a multithread, static version of the CRT.

- An app that's built by using the /MDd compiler option.

  That is, a debug, multithread, and DLL-specific version of the CRT. Such an app is not supported on the Windows Runtime.

For a complete list of CRT functions that are not available in a UWP app and suggestions for alternative functions, see CRT functions not supported in Universal Windows Platform apps.

See also

Compatibility
Windows Runtime Unsupported CRT Functions
Universal C runtime routines by category
Create a Universal Windows Platform console app

ANSI C Compliance

3/11/2019 • 2 minutes to read • Edit Online

The naming convention for all Microsoft-specific identifiers in the run-time system (such as functions, macros, constants, variables, and type definitions) is ANSI-compliant. In this documentation, any run-time function that follows the ANSI/ISO C standards is noted as being ANSI compatible. ANSI-compliant applications should only use these ANSI compatible functions.

The names of Microsoft-specific functions and global variables begin with a single underscore. These names can be overridden only locally, within the scope of your code. For example, when you include Microsoft run-time header files, you can still locally override the Microsoft-specific function named `_open` by declaring a local variable of the same name. However, you cannot use this name for your own global function or global variable.

The names of Microsoft-specific macros and manifest constants begin with two underscores, or with a single leading underscore immediately followed by an uppercase letter. The scope of these identifiers is absolute. For example, you cannot use the Microsoft-specific identifier **_UPPER** for this reason.

See also

Compatibility

UNIX

3/11/2019 • 2 minutes to read • Edit Online

If you plan to port your programs to UNIX, follow these guidelines:

- Do not remove header files from the SYS subdirectory. You can place the SYS header files elsewhere only if you do not plan to transport your programs to UNIX.
- Use the UNIX-compatible path delimiter in routines that take strings representing paths and filenames as arguments. UNIX supports only the forward slash (/) for this purpose, whereas Win32 operating systems support both the backslash (\) and the forward slash (/). Thus this documentation uses UNIX-compatible forward slashes as path delimiters in `#include` statements, for example. (However, the Windows operating system command shell, CMD.EXE, does not support the forward slash in commands entered at the command prompt.)
- Use paths and filenames that work correctly in UNIX, which is case sensitive. The file allocation table (FAT) file system in Win32 operating systems is not case sensitive; the NTFS file system preserves case for directory listings but ignores case in file searches and other system operations.

> **NOTE**
> In this version of Visual C++, UNIX compatibility information has been removed from the function descriptions.

See also

Compatibility

Windows Platforms (CRT)

10/31/2018 • 2 minutes to read • Edit Online

The C run-time libraries for Visual Studio support current versions of Windows and Windows Server, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008, and Windows Vista, and optionally support Windows XP Service Pack 3 (SP3) for x86, Windows XP Service Pack 2 (SP2) for x64, and Windows Server 2003 Service Pack 2 (SP2) for both x86 and x64. All of these operating systems support the Windows desktop API (Win32) and provide Unicode support. In addition, any Win32 application can use a multibyte character set (MBCS).

> **NOTE**
>
> The default installation of the **Desktop development with C++** workload in Visual Studio 2017 does not include support for Windows XP and Windows Server 2003 development. You must install the optional component **Windows XP support for C++** to enable a Windows XP platform toolset.

See also

Compatibility

Backward Compatibility

3/11/2019 • 2 minutes to read • Edit Online

For compatibility between product versions, the library OLDNAMES.LIB maps old names to new names. For instance, `open` maps to `_open`. You must explicitly link with OLDNAMES.LIB only when you compile with the following combinations of command-line options:

- `/Zl` (omit default library name from object file) and `/Ze` (the default — use Microsoft extensions)
- `/link` (linker-control), `/NOD` (no default-library search), and `/Ze`

For more information about compiler command-line options, see Compiler Reference.

See also

Compatibility

Required and Optional Header Files

3/11/2019 • 2 minutes to read • Edit Online

The description of each run-time routine includes a list of the required and optional include, or header (.H), files for that routine. Required header files need to be included to obtain the function declaration for the routine or a definition used by another routine called internally. Optional header files are usually included to take advantage of predefined constants, type definitions, or inline macros. The following table lists some examples of optional header file contents:

DEFINITION	EXAMPLE
Macro definition	If a library routine is implemented as a macro, the macro definition may be in a header file other than the header file for the original routine. For instance, the `_toupper` macro is defined in the header file CTYPE.H, while the function `toupper` is declared in STDLIB.H.
Predefined Constant	Many library routines refer to constants that are defined in header files. For instance, the `_open` routine uses constants such as `_O_CREAT`, which is defined in the header file FCNTL.H.
Type definition	Some library routines return a structure or take a structure as an argument. For example, stream input/output routines use a structure of type `FILE`, which is defined in STDIO.H.

The run-time library header files provide function declarations in the ANSI/ISO C standard recommended style. The compiler performs type checking on any routine reference that occurs after its associated function declaration. Function declarations are especially important for routines that return a value of some type other than `int`, which is the default. Routines that do not specify their appropriate return value in their declaration will be considered by the compiler to return an `int`, which can cause unexpected results. See Type Checking for more information.

See also

CRT Library Features

Files and Streams

3/11/2019 • 2 minutes to read • Edit Online

A program communicates with the target environment by reading and writing files. A file can be:

- A data set that you can read and write repeatedly.
- A stream of bytes generated by a program (such as a pipeline).
- A stream of bytes received from or sent to a peripheral device.

The last two items are interactive files. Files are typically the principal means by which to interact with a program. You manipulate all these kinds of files in much the same way — by calling library functions. You include the standard header STDIO.H to declare most of these functions.

Before you can perform many of the operations on a file, the file must be opened. Opening a file associates it with a stream, a data structure within the Standard C Library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type FILE.

The target environment opens three files before program startup. You can open a file by calling the library function fopen, _wfopen with two arguments. (The `fopen` function has been deprecated, use fopen_s, _wfopen_s instead.) The first argument is a filename. The second argument is a C string that specifies:

- Whether you intend to read data from the file or write data to it or both.
- Whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place.
- Whether writes to a file can alter existing contents or should only append bytes at the end of the file.
- Whether you want to manipulate a text stream or a binary stream.

Once the file is successfully opened, you can then determine whether the stream is byte oriented (a byte stream) or wide oriented (a wide stream). A stream is initially unbound. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to fclose or freopen.

See also

Text and Binary Streams
Byte and Wide Streams
Controlling Streams
Stream States

Text and Binary Streams

3/11/2019 • 2 minutes to read • Edit Online

A text stream consists of one or more lines of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an `NL` (newline) at the end of each line. When writing to a text stream, the program writes an `NL` to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling fgetpos or ftell. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling fsetpos or fseek. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- Empty files.
- Space characters at the end of a line.
- Partial lines (by omitting the `NL` at the end of a file).
- characters other than the printable characters, NL, and `HT` (horizontal tab).

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A binary stream consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by ftell and fseek count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

A byte stream treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

See also

Files and Streams

Byte and Wide Streams

3/11/2019 • 2 minutes to read • Edit Online

A byte stream treats a file as a sequence of bytes. Within the program, the stream is the identical sequence of bytes.

By contrast, a wide stream treats a file as a sequence of generalized multibyte characters, which can have a broad range of encoding rules. (Text and binary files are still read and written as previously described.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C Library. The conversion rules can, in principle, be altered by a call to setlocale that alters the category `LC_CTYPE`. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category `LC_CTYPE` subsequently changes.

Positioning within a wide stream suffers the same limitations as for text steams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type `mbstate_t`. Thus, the only reliable way to obtain a file position within a wide stream is by calling fgetpos, and the only reliable way to restore a position obtained this way is by calling fsetpos.

See also

Files and Streams
setlocale, _wsetlocale

Controlling Streams

3/11/2019 • 2 minutes to read • Edit Online

fopen returns the address of an object of type `FILE`. You use this address as the `stream` argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling fgetc, and all output takes place as if each character is written by calling fputc. For a wide stream, all input takes place as if each character is read by calling fgetwc, and all output takes place as if each character is written by calling fputwc.

You can close a file by calling fclose, after which the address of the `FILE` object is invalid.

A `FILE` object stores the state of a stream, including:

- An error indicator set nonzero by a function that encounters a read or write error.
- An end-of-file indicator set nonzero by a function that encounters the end of the file while reading.
- A file-position indicator specifies the next byte in the stream to read or write, if the file can support positioning requests.
- A stream state specifies whether the stream will accept reads and/or writes and whether the stream is unbound, byte oriented, or wide oriented.
- A conversion state remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file).
- A file buffer specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream.

Do not alter any value stored in a `FILE` object or in a file buffer that you specify for use with that object. You cannot copy a `FILE` object and portably use the address of the copy as a `stream` argument to a library function.

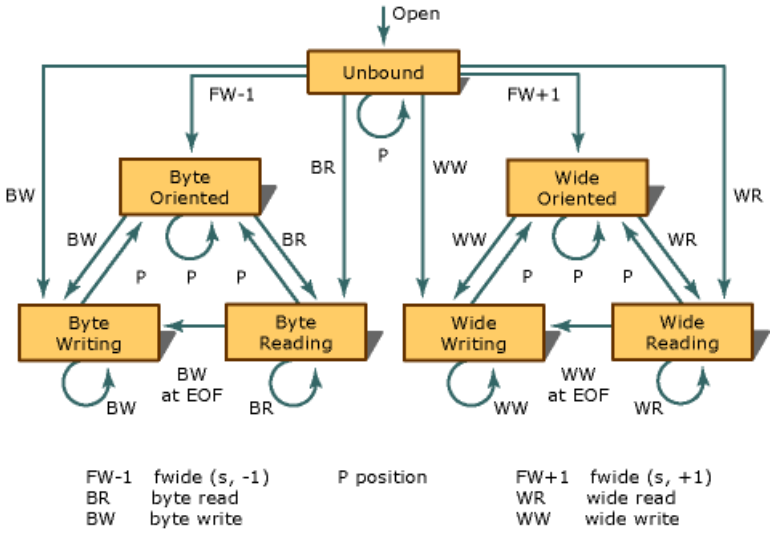
See also

Files and Streams

Stream States

3/11/2019 • 2 minutes to read • Edit Online

The valid states, and state transitions, for a stream are shown in the following figure.



Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in <stdio.h>:

- The byte read functions — fgetc, fgets, fread, fscanf, getc, getchar, gets, scanf, and ungetc
- The byte write functions — fprintf, fputc, fputs, fwrite, printf, putc, putchar, puts, vfprintf, and vprintf
- The position functions — fflush, fseek, fsetpos, and rewind

Functions in the remaining two groups are declared in <wchar.h>:

- The wide read functions — fgetwc, fgetws, fwscanf, getwc, getwchar, ungetwc, and wscanf,
- The wide write functions — fwprintf, fputwc, fputws, putwc, putwchar, vfwprintf, vwprintf, and wprintf,

The state diagram shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never decreases the number of valid function calls that can follow.

See also

Files and Streams

Recommendations for Choosing Between Functions and Macros

3/11/2019 • 2 minutes to read • Edit Online

Most Microsoft run-time library routines are compiled or assembled functions, but some routines are implemented as macros. When a header file declares both a function and a macro version of a routine, the macro definition takes precedence, because it always appears after the function declaration. When you invoke a routine that is implemented as both a function and a macro, you can force the compiler to use the function version in two ways:

- Enclose the routine name in parentheses.

```
#include <ctype.h>
a = _toupper(a);    // Use macro version of toupper.
a = (_toupper)(a);  // Force compiler to use
                    // function version of toupper.
```

- "Undefine" the macro definition with the `#undef` directive:

```
#include <ctype.h>
#undef _toupper
```

If you need to choose between a function and a macro implementation of a library routine, consider the following trade-offs:

- **Speed versus size** The main benefit of using macros is faster execution time. During preprocessing, a macro is expanded (replaced by its definition) inline each time it is used. A function definition occurs only once regardless of how many times it is called. Macros may increase code size but do not have the overhead associated with function calls.
- **Function evaluation** A function evaluates to an address; a macro does not. Thus you cannot use a macro name in contexts requiring a pointer. For instance, you can declare a pointer to a function, but not a pointer to a macro.
- **Type-checking** When you declare a function, the compiler can check the argument types. Because you cannot declare a macro, the compiler cannot check macro argument types; although it can check the number of arguments you pass to a macro.

See also

CRT Library Features

Type Checking (CRT)

3/11/2019 • 2 minutes to read • Edit Online

The compiler performs limited type checking on functions that can take a variable number of arguments, as follows:

FUNCTION CALL	TYPE-CHECKED ARGUMENTS
`_cprintf_s`, `_cscanf_s`, `printf_s`, `scanf_s`	First argument (format string)
`fprintf_s`, `fscanf_s`, `sprintf_s`, `sscanf_s`	First two arguments (file or buffer and format string)
`_snprintf_s`	First three arguments (file or buffer, count, and format string)
`_open`	First two arguments (path and `_open` flag)
`_sopen_s`	First three arguments (path, `_open` flag, and sharing mode)
`_execl`, `_execle`, `_execlp`, `_execlpe`	First two arguments (path and first argument pointer)
`_spawnl`, `_spawnle`, `_spawnlp`, `_spawnlpe`	First three arguments (mode flag, path, and first argument pointer)

The compiler performs the same limited type checking on the wide-character counterparts of these functions.

See also

CRT Library Features

Direction Flag

3/11/2019 • 2 minutes to read • Edit Online

The direction flag is a CPU flag specific to all Intel x86-compatible CPUs. It applies to all assembly instructions that use the REP (repeat) prefix, such as MOVS, MOVSD, MOVSW, and others. Addresses provided to applicable instructions are increased if the direction flag is cleared.

The C run-time routines assume that the direction flag is cleared. If you are using other functions with the C run-time functions, you must ensure that the other functions leave the direction flag alone or restore it to its original condition. Expecting the direction flag to be clear upon entry makes the run-time code faster and more efficient.

The C Run-Time library functions, such as the string-manipulation and buffer-manipulation routines, expect the direction flag to be clear.

See also

CRT Library Features

Security Features in the CRT

3/11/2019 • 3 minutes to read • Edit Online

Many old CRT functions have newer, more secure versions. If a secure function exists, the older, less secure version is marked as deprecated and the new version has the `_s` ("secure") suffix.

In this context, "deprecated" just means that a function's use is not recommended; it does not indicate that the function is scheduled to be removed from the CRT.

The secure functions do not prevent or correct security errors; rather, they catch errors when they occur. They perform additional checks for error conditions, and in the case of an error, they invoke an error handler (see Parameter Validation).

For example, the `strcpy` function has no way of telling if the string that it is copying is too big for its destination buffer. However, its secure counterpart, `strcpy_s`, takes the size of the buffer as a parameter, so it can determine if a buffer overrun will occur. If you use `strcpy_s` to copy eleven characters into a ten-character buffer, that is an error on your part; `strcpy_s` cannot correct your mistake, but it can detect your error and inform you by invoking the invalid parameter handler.

Eliminating deprecation warnings

There are several ways to eliminate deprecation warnings for the older, less secure functions. The simplest is simply to define `_CRT_SECURE_NO_WARNINGS` or use the warning pragma. Either will disable deprecation warnings, but of course the security issues that caused the warnings still exist. It is far better to leave deprecation warnings enabled and take advantage of the new CRT security features.

In C++, the easiest way to do that is to use Secure Template Overloads, which in many cases will eliminate deprecation warnings by replacing calls to deprecated functions with calls to the new secure versions of those functions. For example, consider this deprecated call to `strcpy`:

```
char szBuf[10];
strcpy(szBuf, "test"); // warning: deprecated
```

Defining `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` as 1 eliminates the warning by changing the `strcpy` call to `strcpy_s`, which prevents buffer overruns. For more information, see Secure Template Overloads.

For those deprecated functions without secure template overloads, you should definitely consider manually updating your code to use the secure versions.

Another source of deprecation warnings, unrelated to security, is the POSIX functions. Replace POSIX function names with their standard equivalents (for example, change access to _access), or disable POSIX-related deprecation warnings by defining `_CRT_NONSTDC_NO_WARNINGS`. For more information, see Compatibility.

Additional Security Features

Some of the security features include the following:

- `Parameter Validation`. Parameters passed to CRT functions are validated, in both secure functions and in many preexisting versions of functions. These validations include:

    - Checking for **NULL** values passed to the functions.
    - Checking enumerated values for validity.
    - Checking that integral values are in valid ranges.
- For more information, see Parameter Validation.
- A handler for invalid parameters is also accessible to the developer. When an encountering an invalid parameter, instead of asserting and exiting the application, the CRT provides a way to check these problems with the _set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler function.
- `Sized Buffers`. The secure functions require that the buffer size be passed to any function that writes to a buffer. The secure versions validate that the buffer is large enough before writing to it, helping to avoid dangerous buffer overrun errors that could allow malicious code to execute. These functions usually return an `errno` type of error code and invoke the invalid parameter handler if the size of the buffer is too small. Functions that read from input buffers, such as `gets`, have secure versions that require you to specify a maximum size.
- `Null termination`. Some functions that left potentially non-terminated strings have secure versions which ensure that strings are properly null-terminated.
- `Enhanced error reporting`. The secure functions return error codes with more error information than was available with the preexisting functions. The secure functions and many of the preexisting functions now set `errno` and often return an `errno` code type as well, to provide better error reporting.
- `Filesystem security`. Secure file I/O APIs support secure file access in the default case.
- `Windows security`. Secure process APIs enforce security policies and allow ACLs to be specified.
- `Format string syntax checking`. Invalid strings are detected, for example, using incorrect type field characters in `printf` format strings.

See also

Parameter Validation
Secure Template Overloads
CRT Library Features

Security-Enhanced Versions of CRT Functions

3/11/2019 • 5 minutes to read • Edit Online

More secure versions of run-time library routines are available. For further information concerning security enhancements in the CRT, see Security Features in the CRT.

Secure functions

CRT FUNCTION	SECURITY ENHANCED FUNCTION	USE
_access, _waccess	_access_s, _waccess_s	Determine file-access permission
_alloca	_malloca	Allocate memory on the stack
asctime, _wasctime	asctime_s, _wasctime_s	Convert time from type `struct tm` to character string
bsearch	bsearch_s	Perform a binary search of a sorted array
_cgets, _cgetws	_cgets_s, _cgetws_s	Get a character string from the console
_chsize	_chsize_s	Change the size of a file
clearerr	clearerr_s	Reset the error indicator for a stream
_control87, _controlfp, __control87_2	_controlfp_s	Get and set the floating-point control word
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l	_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l	Format and print to the console
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l	_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l	Read formatted data from the console
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64	_ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s	Convert time from type `time_t`, `__time32_t` or `__time64_t` to character string
_ecvt	_ecvt_s	Convert a `double` number to a string
_fcvt	_fcvt_s	Converts a floating-point number to a string
fopen, _wfopen	fopen_s, _wfopen_s	Open a file
fprintf, _fprintf_l, fwprintf, _fwprintf_l	fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l	Print formatted data to a stream
fread	fread_s	Read from a file

CRT FUNCTION	SECURITY ENHANCED FUNCTION	USE
_fread_nolock	_fread_nolock_s	Read from a file without using a multi-thread write lock
freopen, _wfreopen	freopen_s, _wfreopen_s	Reopen the file
fscanf, _fscanf_l, fwscanf, _fwscanf_l	fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l	Read formatted data from a stream
_ftime, _ftime32, _ftime64	_ftime_s, _ftime32_s, _ftime64_s	Get the current time
_gcvt	_gcvt_s	Convert a floating-point value to a string, and store it in a buffer
getenv, _wgetenv	getenv_s, _wgetenv_s	Get a value from the current environment.
gets, getws	gets_s, _getws_s	Get a line from the `stdin` stream
gmtime, _gmtime32, _gmtime64	_gmtime32_s, _gmtime64_s	Convert time from type `time_t` to `struct tm` or from type `__time64_t` to `struct tm`
itoa, _itoa, ltoa, _ltoa, ultoa, _ultoa, _i64toa, _ui64toa, _itow, _ltow, _ultow, _i64tow, _ui64tow	_itoa_s, _ltoa_s, _ultoa_s, _i64toa_s, _ui64toa_s, _itow_s, _ltow_s, _ultow_s, _i64tow_s, _ui64tow_s	Convert an integral type to a string
_lfind	_lfind_s	Perform a linear search for the specified key
localtime, _localtime32, _localtime64	localtime_s, _localtime32_s, _localtime64_s	Convert time from type `time_t` to `struct tm` or from type `__time64_t` to `struct tm` with local correction
_lsearch	_lsearch_s	Perform a linear search for a value; adds to end of list if not found
_makepath, _wmakepath	_makepath_s, _wmakepath_s	Create a path name from components
_mbccpy, _mbccpy_l	_mbccpy_s, _mbccpy_s_l	Copy a multibyte character from one string to another string
_mbsnbcat, _mbsnbcat_l	_mbsnbcat_s, _mbsnbcat_s_l	Append, at most, the first *n* bytes of one multibyte character string to another
_mbsnbcpy, _mbsnbcpy_l	_mbsnbcpy_s, _mbsnbcpy_s_l	Copy *n* bytes of a string to a destination string
_mbsnbset, _mbsnbset_l	_mbsnbset_s, _mbsnbset_s_l	Set the first *n* bytes of a string to a specified character
mbsrtowcs	mbsrtowcs_s	Convert a multibyte character string to a corresponding wide character string

CRT FUNCTION	SECURITY ENHANCED FUNCTION	USE
mbstowcs, _mbstowcs_l	mbstowcs_s, _mbstowcs_s_l	Convert a sequence of multibyte characters to a corresponding sequence of wide characters
memcpy, wmemcpy	memcpy_s, wmemcpy_s	Copy characters between buffers
memmove, wmemmove	memmove_s, wmemmove_s	Move one buffer to another
_mktemp, _wmktemp	_mktemp_s, _wmktemp_s	Create a unique filename
printf, _printf_l, wprintf, _wprintf_l	printf_s, _printf_s_l, wprintf_s, _wprintf_s_l	Print formatted output to the standard output stream
_putenv, _wputenv	_putenv_s, _wputenv_s	Create, modify, or remove environment variables
qsort	qsort_s	Perform a quick sort
rand	rand_s	Generate a pseudorandom number
scanf, _scanf_l, wscanf, _wscanf_l	scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l	Read formatted data from the standard input stream
_searchenv, _wsearchenv	_searchenv_s, _wsearchenv_s	Search for a file using environment paths
snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l	_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l	Write formatted data to a string
_snscanf, _snscanf_l, _snwscanf, _snwscanf_l	_snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from a string.
_sopen, _wsopen	_sopen_s, _wsopen_s	Open a file for sharing
_splitpath, _wsplitpath	_splitpath_s, _wsplitpath_s	Break a path name into components
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l	sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l	Write formatted data to a string
sscanf, _sscanf_l, swscanf, _swscanf_l	sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data from a string
strcat, wcscat, _mbscat	strcat_s, wcscat_s, _mbscat_s	Append a string
strcpy, wcscpy, _mbscpy	strcpy_s, wcscpy_s, _mbscpy_s	Copy a string
_strdate, _wstrdate	_strdate_s, _wstrdate_s	Return current system date as string
strerror, _strerror, _wcserror, __wcserror	strerror_s, _strerror_s, _wcserror_s, __wcserror_s	Get a system error message (`strerror` , `_wcserror`) or print a user-supplied error message (`_strerror` , `__wcserror`)

CRT FUNCTION	SECURITY ENHANCED FUNCTION	USE
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l	_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l	Convert a string to lowercase
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l	strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l	Append characters to a string
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l	strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l	Copy characters of one string to another
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l	_strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l	Set the first n characters of a string to the specified character
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l	_strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l	Set all the characters of a string to the specified character
_strtime, _wstrtime	_strtime_s, _wstrtime_s	Return current system time as string
strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l	strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l	Find the next token in a string, using the current locale or a locale passed in
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr	_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert a string to uppercase
tmpfile	tmpfile_s	Create a temporary file
_tempnam, _wtempnam, tmpnam, _wtmpnam	tmpnam_s, _wtmpnam_s	Generate names you can use to create temporary files
_umask	_umask_s	Set the default file-permission mask
_vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l	_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l	Write formatted output to the console using a pointer to a list of arguments
vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l	vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l	Write formatted output using a pointer to a list of arguments
vfscanf, vfwscanf	vfscanf_s, vfwscanf_s	Read formatted data from a stream
vprintf, _vprintf_l, vwprintf, _vwprintf_l	vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	Write formatted output using a pointer to a list of arguments
vscanf, vwscanf	vscanf_s, vwscanf_s	Read formatted data from the standard input stream
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l	vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted output using a pointer to a list of arguments
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l	vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l	Write formatted output using a pointer to a list of arguments
vsscanf, vswscanf	vsscanf_s, vswscanf_s	Read formatted data from a string

CRT FUNCTION	SECURITY ENHANCED FUNCTION	USE
wcrtomb	wcrtomb_s	Convert a wide character into its multibyte character representation
wcsrtombs	wcsrtombs_s	Convert a wide character string to its multibyte character string representation
wcstombs, _wcstombs_l	wcstombs_s, _wcstombs_s_l	Convert a sequence of wide characters to a corresponding sequence of multibyte characters
wctomb, _wctomb_l	wctomb_s, _wctomb_s_l	Convert a wide character to the corresponding multibyte character

See also

CRT Library Features

Parameter Validation

3/11/2019 • 2 minutes to read • Edit Online

Most of the security-enhanced CRT functions and many of the preexisting functions validate their parameters. This could include checking pointers for **NULL**, checking that integers fall into a valid range, or checking that enumeration values are valid. When an invalid parameter is found, the invalid parameter handler is executed.

Invalid Parameter Handler Routine

When a C Runtime Library function detects an invalid parameter, it captures some information about the error, and then calls a macro that wraps an invalid parameter handler dispatch function, one of _invalid_parameter, _invalid_parameter_noinfo, or _invalid_parameter_noinfo_noreturn. The dispatch function called depends on whether your code is, respectively, a debug build, a retail build, or the error is not considered recoverable.

In Debug builds, the invalid parameter macro usually raises a failed assertion and a debugger breakpoint before the dispatch function is called. When the code is executed, the assertion may be reported to the user in a dialog box that has "Abort", "Retry", and "Continue" or similar choices, depending on the operating system and runtime library version. These options allow the user to immediately terminate the program, to attach a debugger, or to let the existing code continue to run, which calls the dispatch function.

The invalid parameter handler dispatch function in turn calls the currently assigned invalid parameter handler. By default, the invalid parameter calls `_invoke_watson` which causes the application to "crash," that is, terminate and generate a mini-dump. If enabled by the operating system, a dialog box asks the user if they want to load the crash dump to Microsoft for analysis.

This behavior can be changed by using the functions _set_invalid_parameter_handler or _set_thread_local_invalid_parameter_handler to set the invalid parameter handler to your own function. If the function you specify does not terminate the application, control is returned to the function that received the invalid parameters. In the CRT, these functions will normally cease function execution, set `errno` to an error code, and return an error code. In many cases, the `errno` value and the return value are both `EINVAL`, indicating an invalid parameter. In some cases, a more specific error code is returned, such as `EBADF` for a bad file pointer passed in as a parameter. For more information on `errno`, see errno, _doserrno, _sys_errlist, and _sys_nerr.

See also

Secure Template Overloads

3/11/2019 • 2 minutes to read • Edit Online

Microsoft has deprecated many C Runtime library (CRT) functions in favor of security-enhanced versions. For example, `strcpy_s` is the more secure replacement for `strcpy`. The deprecated functions are common sources of security bugs, because they do not prevent operations that can overwrite memory. By default, the compiler produces a deprecation warning when you use one of these functions. The CRT provides C++ template overloads for these functions to help ease the transition to the more secure variants.

For example, this code snippet generates a warning because `strcpy` is deprecated:

```
char szBuf[10];
strcpy(szBuf, "test"); // warning: deprecated
```

The deprecation warning is there to tell you that your code may be unsafe. If you have verified that your code can't overwrite memory, you have several choices. You can choose to ignore the warning, you can define the symbol `_CRT_SECURE_NO_WARNINGS` before the include statements for the CRT headers to suppress the warning, or you can update your code to use `strcpy_s`:

```
char szBuf[10];
strcpy_s(szBuf, 10, "test"); // security-enhanced _s function
```

The template overloads provide additional choices. If you define `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` to 1, this enables template overloads of standard CRT functions that call the more secure variants automatically. If `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` is 1, then no changes to your code are necessary. Behind the scenes, the call to `strcpy` is changed to a call to `strcpy_s` with the size argument supplied automatically.

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1

// ...

char szBuf[10];
strcpy(szBuf, "test"); // ==> strcpy_s(szBuf, 10, "test")
```

The macro `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` does not affect the functions that take a count, such as `strncpy`. To enable template overloads for the count functions, define `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT` to 1. Before doing so, however, make sure that your code passes the count of characters, not the size of the buffer (a common mistake). Also, code that explicitly writes a null terminator at the end of the buffer after the function call is unnecessary if the secure variant is called. If you need truncation behavior, see _TRUNCATE.

> **NOTE**
> The macro `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT` requires that `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` is also defined as 1. If `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT` is defined as 1 and `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` is defined as 0, the application will not perform any template overloads.

When you define `_CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES` to 1, it enables template overloads of the secure variants (names ending in "_s"). In this case, if `_CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES` is 1, then one small change must be made to the original code:

```
#define _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES 1

// ...

char szBuf[10];
strcpy_s(szBuf, "test"); // ==> strcpy_s(szBuf, 10, "test")
```

Only the name of the function needs to be changed (by adding "_s"); the template overload takes care of providing the size argument.

By default, `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES` and `_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT` are defined as 0 (disabled) and `_CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES` is defined as 1 (enabled).

Note that these template overloads only work for static arrays. Dynamically allocated buffers require additional source code changes. Revisiting the above examples:

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1

// ...

char *szBuf = (char*)malloc(10);
strcpy(szBuf, "test"); // still deprecated; you have to change it to
                       // strcpy_s(szBuf, 10, "test");
```

And this:

```
#define _CRT_SECURE_CPP_OVERLOAD_SECURE_NAMES 1

// ...

char *szBuf = (char*)malloc(10);
strcpy_s(szBuf, "test"); // doesn't compile; you have to change it to
                         // strcpy_s(szBuf, 10, "test");
```

See also

Security Features in the CRT
CRT Library Features

SAL Annotations

3/11/2019 • 2 minutes to read • Edit Online

If you examine the library header files, you may notice some unusual annotations, for example, `_In_z` and `_Out_z_cap_(_Size)`. These are examples of the Microsoft source-code annotation language (SAL), which provides a set of annotations to describe how a function uses its parameters, for example, the assumptions it makes about them and the guarantees it makes on finishing. The header file <sal.h> defines the annotations.

For more information about using SAL annotations in Visual Studio, see Using SAL Annotations to Reduce C/C++ Code Defects.

See also

CRT Library Features

Multithreaded Libraries Performance

3/11/2019 • 2 minutes to read • Edit Online

The single-threaded CRT is no longer available. This topic discusses how to get the maximum performance from the multithreaded libraries.

Maximizing performance

The performance of the multithreaded libraries has been improved and is close to the performance of the now-eliminated single-threaded libraries. For those situations when even higher performance is required, there are several new features.

- Independent stream locking allows you to lock a stream and then use _nolock Functions that access the stream directly. This allows lock usage to be hoisted outside critical loops.
- Per-thread locale reduces the cost of locale access for multithreaded scenarios (see _configthreadlocale).
- Locale-dependent functions (with names ending in _l) take the locale as a parameter, removing substantial cost (for example, printf, _printf_l, wprintf, _wprintf_l).
- Optimizations for common codepages reduce the cost of many short operations.
- Defining _CRT_DISABLE_PERFCRIT_LOCKS forces all I/O operations to assume a single-threaded I/O model and use the _nolock forms of the functions. This allows highly I/O-based single-threaded applications to get better performance.
- Exposure of the CRT heap handle allows you to enable the Windows Low Fragmentation Heap (LFH) for the CRT heap, which can substantially improve performance in highly scaled scenarios.

See also

CRT Library Features

Link Options

10/31/2018 • 2 minutes to read • Edit Online

The CRT lib directory includes a number of small object files that enable specific CRT features without any code change. These are called "link options" since you just have to add them to the linker command line to use them.

CLR pure mode versions of these objects are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017. Use the regular versions for native and /clr code.

NATIVE AND /CLR	PURE MODE	DESCRIPTION
binmode.obj	pbinmode.obj	Sets the default file-translation mode to binary. See _fmode.
chkstk.obj	n/a	Provides stack-checking and alloca support when not using the CRT.
commode.obj	pcommode.obj	Sets the global commit flag to "commit". See fopen, _wfopen and fopen_s, _wfopen_s.
exe_initialize_mta.lib	n/a	Initializes the MTA apartment during EXE startup, which allows the use of COM objects in global smart pointers. Because this option leaks an MTA apartment reference during shutdown, do not use it for DLLs. Linking to this is equivalent to including combase.h and defining _EXE_INITIALIZE_MTA.
fp10.obj	n/a	Changes the default precision control to 64 bits. See Floating-Point Support.
invalidcontinue.obj	pinvalidcontinue.obj	Sets a default invalid parameter handler that does nothing, meaning that invalid parameters passed to CRT functions will just set errno and return an error result.
loosefpmath.obj	n/a	Ensures that floating point code tolerates denormal values.
newmode.obj	pnewmode.obj	Causes malloc to call the new handler on failure. See _set_new_mode, _set_new_handler, calloc, and realloc.
noarg.obj	pnoarg.obj	Disables all processing of argc and argv.
nochkclr.obj	n/a	Does nothing. Remove from your project.
noenv.obj	pnoenv.obj	Disables the creation of a cached environment for the CRT.

NATIVE AND /CLR	PURE MODE	DESCRIPTION
nothrownew.obj	pnothrownew.obj	Enables the non-throwing version of new in the CRT. See new and delete Operators.
setargv.obj	psetargv.obj	Enables command-line argument wildcard expansion. See Expanding Wildcard Arguments.
threadlocale.obj	pthreadlocale.obj	Enables per-thread locale for all new threads by default.
wsetargv.obj	pwsetargv.obj	Enables command-line argument wildcard expansion. See Expanding Wildcard Arguments.

See also

- CRT Library Features

Potential Errors Passing CRT Objects Across DLL Boundaries

5/8/2019 • 3 minutes to read • Edit Online

When you pass C Run-time (CRT) objects such as file handles, locales, and environment variables into or out of a DLL (function calls across the DLL boundary), unexpected behavior can occur if the DLL, as well as the files calling into the DLL, use different copies of the CRT libraries.

A related problem can occur when you allocate memory (either explicitly with `new` or `malloc`, or implicitly with `strdup`, `strstreambuf::str`, and so on) and then pass a pointer across a DLL boundary to be freed. This can cause a memory access violation or heap corruption if the DLL and its users use different copies of the CRT libraries.

Another symptom of this problem can be an error in the output window during debugging such as:

HEAP[]: Invalid Address specified to RtlValidateHeap(#,#)

Causes

Each copy of the CRT library has a separate and distinct state, kept in thread local storage by your app or DLL. As such, CRT objects such as file handles, environment variables, and locales are only valid for the copy of the CRT in the app or DLL where these objects are allocated or set. When a DLL and its app clients use different copies of the CRT library, you cannot pass these CRT objects across the DLL boundary and expect them to be picked up correctly on the other side. This is particularly true of CRT versions before the Universal CRT in Visual Studio 2015 and later. There was a version-specific CRT library for every version of Visual Studio built with Visual Studio 2013 or earlier. Internal implementation details of the CRT, for example, its data structures and naming conventions, were different in each version. Dynamically linking code compiled for one version of the CRT to a different version of the CRT DLL has never been supported, though occasionally it would work, more by luck than by design.

Also, because each copy of the CRT library has its own heap manager, allocating memory in one CRT library and passing the pointer across a DLL boundary to be freed by a different copy of the CRT library is a potential cause for heap corruption. If you design your DLL so that it passes CRT objects across the boundary or allocates memory and expects it to be freed outside the DLL, you restrict the app clients of the DLL to use the same copy of the CRT library as the DLL. The DLL and its clients normally use the same copy of the CRT library only if both are linked at load time to the same version of the CRT DLL. Because the DLL version of the Universal CRT library used by Visual Studio 2015 and later on Windows 10 is now a centrally deployed Windows component, ucrtbase.dll, it is the same for apps built with Visual Studio 2015 and later versions. However, even when the CRT code is identical, you can't hand off memory allocated in one heap to a component that uses a different heap.

Example

Description

This example passes a file handle across a DLL boundary.

The DLL and .exe file are built with /MD, so they share a single copy of the CRT.

If you rebuild with /MT so that they use separate copies of the CRT, running the resulting test1Main.exe results in an access violation.

```
// test1Dll.cpp
// compile with: cl /EHsc /W4 /MD /LD test1Dll.cpp
#include <stdio.h>
__declspec(dllexport) void writeFile(FILE *stream)
{
   char   s[] = "this is a string\n";
   fprintf( stream, "%s", s );
   fclose( stream );
}
```

```
// test1Main.cpp
// compile with: cl /EHsc /W4 /MD test1Main.cpp test1Dll.lib
#include <stdio.h>
#include <process.h>
void writeFile(FILE *stream);

int main(void)
{
   FILE  * stream;
   errno_t err = fopen_s( &stream, "fprintf.out", "w" );
   writeFile(stream);
   system( "type fprintf.out" );
}
```

```
this is a string
```

Example

Description

This example passes environment variables across a DLL boundary.

```
// test2Dll.cpp
// compile with: cl /EHsc /W4 /MT /LD test2Dll.cpp
#include <stdio.h>
#include <stdlib.h>

__declspec(dllexport) void readEnv()
{
   char *libvar;
   size_t libvarsize;

   /* Get the value of the MYLIB environment variable. */
   _dupenv_s( &libvar, &libvarsize, "MYLIB" );

   if( libvar != NULL )
      printf( "New MYLIB variable is: %s\n", libvar);
   else
      printf( "MYLIB has not been set.\n");
   free( libvar );
}
```

```
// test2Main.cpp
// compile with: cl /EHsc /W4 /MT test2Main.cpp test2dll.lib
#include <stdlib.h>
#include <stdio.h>

void readEnv();

int main( void )
{
   _putenv( "MYLIB=c:\\mylib;c:\\yourlib" );
   readEnv();
}
```

```
MYLIB has not been set.
```

If both the DLL and .exe file are built with /MD so that only one copy of the CRT is used, the program runs successfully and produces the following output:

```
New MYLIB variable is: c:\mylib;c:\yourlib
```

See also

CRT Library Features

CRT Initialization

5/8/2019 • 2 minutes to read • Edit Online

This topic describes how the CRT initializes global states in native code.

By default, the linker includes the CRT library, which provides its own startup code. This startup code initializes the CRT library, calls global initializers, and then calls the user-provided `main` function for console applications.

Initializing a Global Object

Consider the following code:

```
int func(void)
{
    return 3;
}

int gi = func();

int main()
{
    return gi;
}
```

According to the C/C++ standard, `func()` must be called before `main()` is executed. But who calls it?

One way to determine this is to set a breakpoint in `func()`, debug the application, and examine the stack. This is possible because the CRT source code is included with Visual Studio.

When you browse the functions on the stack, you will find that the CRT is looping through a list of function pointers and calling each one as it encounters them. These functions are either similar to `func()` or constructors for class instances.

The CRT obtains the list of function pointers from the Microsoft C++ compiler. When the compiler sees a global initializer, it generates a dynamic initializer in the `.CRT$XCU` section (where `CRT` is the section name and `XCU` is the group name). To obtain a list of those dynamic initializers run the command **dumpbin /all main.obj**, and then search the `.CRT$XCU` section (when main.cpp is compiled as a C++ file, not a C file). It will be similar to the following:

```
SECTION HEADER #6
.CRT$XCU name
       0 physical address
       0 virtual address
       4 size of raw data
     1F2 file pointer to raw data (000001F2 to 000001F5)
     1F6 file pointer to relocation table
       0 file pointer to line numbers
       1 number of relocations
       0 number of line numbers
40300040 flags
         Initialized Data
         4 byte align
         Read Only

RAW DATA #6
  00000000: 00 00 00 00                                      ....

RELOCATIONS #6
                                                 Symbol    Symbol
Offset    Type              Applied To         Index     Name
--------  ----------------  -----------------  --------  ------
00000000  DIR32                     00000000         C  ??__Egi@@YAXXZ (void __cdecl `dynamic initializer for
'gi''(void))
```

The CRT defines two pointers:

- `__xc_a` in `.CRT$XCA`
- `__xc_z` in `.CRT$XCZ`

Both groups do not have any other symbols defined except `__xc_a` and `__xc_z`.

Now, when the linker reads various `.CRT` groups, it combines them in one section and orders them alphabetically. This means that the user-defined global initializers (which the Microsoft C++ compiler puts in `.CRT$XCU`) will always come after `.CRT$XCA` and before `.CRT$XCZ`.

The section will resemble the following:

```
.CRT$XCA
           __xc_a
.CRT$XCU
           Pointer to Global Initializer 1
           Pointer to Global Initializer 2
.CRT$XCZ
           __xc_z
```

So, the CRT library uses both `__xc_a` and `__xc_z` to determine the start and end of the global initializers list because of the way in which they are laid out in memory after the image is loaded.

See also

CRT Library Features

Universal C runtime routines by category

10/31/2018 • 2 minutes to read • Edit Online

This section lists and describes Universal C runtime (UCRT) library routines by category. For reference convenience, some routines are listed in more than one category. Multibyte-character routines and wide-character routines are grouped with single-byte character counterparts, where they exist.

UCRT library routine categories

The main categories of UCRT library routines are:

Argument Access	Buffer Manipulation
Byte Classification	Character Classification
Complex math support	
Data Alignment	Data Conversion
Debug Routines	Directory Control
Error Handling	Exception Handling Routines
File Handling	Floating-Point Support
Input and Output	Internationalization
Memory Allocation	Process and Environment Control
Robustness	Run-Time Error Checking
Searching and Sorting	String Manipulation
System Calls	Time Management

See also

C Run-Time Library Reference

Argument access

10/31/2018 • 2 minutes to read • Edit Online

The **va_arg**, **va_end**, and **va_start** macros provide access to function arguments when the number of arguments is variable. These macros are defined in <stdarg.h> for ANSI/ISO C compatibility and in <varargs.h> for compatibility with UNIX System V.

Argument-access macros

MACRO	USE
va_arg	Retrieve argument from list
va_end	Reset pointer
va_start	Set pointer to beginning of argument list

See also

Universal C runtime routines by category

Buffer manipulation

10/31/2018 • 2 minutes to read • Edit Online

Use these routines to work with areas of memory on a byte-by-byte basis.

Buffer-manipulation routines

ROUTINE	USE
_memccpy	Copy characters from one buffer to another until given character or given number of characters has been copied
memchr, wmemchr	Return pointer to first occurrence, within specified number of characters, of given character in buffer
memcmp, wmemcmp	Compare specified number of characters from two buffers
memcpy, wmemcpy, memcpy_s, wmemcpy_s	Copy specified number of characters from one buffer to another
_memicmp, _memicmp_l	Compare specified number of characters from two buffers without regard to case
memmove, wmemmove,memmove_s, wmemmove_s	Copy specified number of characters from one buffer to another
memset, wmemset	Use given character to initialize specified number of bytes in the buffer
_swab	Swap bytes of data and store them at specified location

When the source and target areas overlap, only **memmove** is guaranteed to copy the full source properly.

See also

Universal C runtime routines by category

Byte classification

10/31/2018 • 2 minutes to read • Edit Online

Each of these routines tests a specified byte of a multibyte character for satisfaction of a condition. Except where specified otherwise, the output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

> **NOTE**
> By definition, the ASCII characters between 0 and 127 are a subset of all multibyte-character sets. For example, the Japanese katakana character set includes ASCII as well as non-ASCII characters.

The predefined constants in the following table are defined in <ctype.h>.

Multibyte-character byte-classification routines

ROUTINE	BYTE TEST CONDITION
isleadbyte, _isleadbyte_l	Lead byte; test result depends on **LC_CTYPE** category setting of current locale
_ismbbalnum, _ismbbalnum_l	**isalnum** || **_ismbbkalnum**
_ismbbalpha, _ismbbalpha_l	**isalpha** || **_ismbbkalnum**
_ismbbgraph, _ismbbgraph_l	Same as **_ismbbprint**, but **_ismbbgraph** does not include the space character (0x20)
_ismbbkalnum, _ismbbkalnum_l	Non-ASCII text symbol other than punctuation. For example, in code page 932 only, **_ismbbkalnum** tests for katakana alphanumeric
_ismbbkana, _ismbbkana_l	Katakana (0xA1 - 0xDF), code page 932 only
_ismbbkprint, _ismbbkprint_l	Non-ASCII text or non-ASCII punctuation symbol. For example, in code page 932 only, **_ismbbkprint** tests for katakana alphanumeric or katakana punctuation (range: 0xA1 - 0xDF).
_ismbbkpunct, _ismbbkpunct_l	Non-ASCII punctuation. For example, in code page 932 only, **_ismbbkpunct** tests for katakana punctuation.
_ismbblead, _ismbblead_l	First byte of multibyte character. For example, in code page 932 only, valid ranges are 0x81 - 0x9F, 0xE0 - 0xFC.
_ismbbprint, _ismbbprint_l	**isprint** || **_ismbbkprint**. **ismbbprint** includes the space character (0x20)

ROUTINE	BYTE TEST CONDITION
_ismbbpunct, _ismbbpunct_l	**ispunct || _ismbbkpunct**
_ismbbtrail, _ismbbtrail_l	Second byte of multibyte character. For example, in code page 932 only, valid ranges are 0x40 - 0x7E, 0x80 - 0xEC.
_ismbslead, _ismbslead_l	Lead byte (in string context)
ismbstrail, _ismbstrail_l	Trail byte (in string context)
_mbbtype, _mbbtype_l	Return byte type based on previous byte
_mbsbtype, _mbsbtype_l	Return type of byte within string
mbsinit	Tracks the state of a multibyte character conversion.

The **MB_LEN_MAX** macro, defined in <limits.h>, expands to the maximum length in bytes that any multibyte character can have. **MB_CUR_MAX**, defined in <stdlib.h>, expands to the maximum length in bytes of any multibyte character in the current locale.

See also

Universal C runtime routines by category

Character Classification

3/11/2019 • 2 minutes to read • Edit Online

Each of these routines tests a specified single-byte character, wide character, or multibyte character for satisfaction of a condition. (By definition, the ASCII character set between 0 and 127 are a subset of all multibyte-character sets. For example, Japanese katakana includes ASCII as well as non-ASCII characters.)

The test conditions are affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Generally these routines execute faster than tests you might write and should be favored over. For example, the following code executes slower than a call to `isalpha(c)`:

```
if ((c >= 'A') && (c <= 'Z')) || ((c >= 'a') && (c <= 'z'))
    return TRUE;
```

Character-Classification Routines

ROUTINE	CHARACTER TEST CONDITION
isalnum, iswalnum, _isalnum_l, _iswalnum_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l	Alphanumeric
_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l	Multibyte alphanumeric
isalpha, iswalpha, _isalpha_l, _iswalpha_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l	Alphabetic
isascii, __isascii, iswascii	ASCII
isblank, iswblank, _isblank_l, _iswblank_l, _ismbcsblank, _ismbcsblank_l	Blank (space or horizontal tab)
iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l	Control
iscsym, iscsymf, __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l	Letter, underscore, or digit
iscsym, iscsymf, __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l	Letter or underscore
isdigit, iswdigit, _isdigit_l, _iswdigit_l, _ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l	Decimal digit

ROUTINE	CHARACTER TEST CONDITION
isgraph, iswgraph, _isgraph_l, _iswgraph_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l	Printable other than space
islower, iswlower, _islower_l, _iswlower_l, _ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Lowercase
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Hiragana
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	Katakana
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Legal multibyte character
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level 0 multibyte character
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level 1 multibyte character
_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l	Japan-level 2 multibyte character
_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l	Non-alphanumeric multibyte character
isprint, iswprint, _isprint_l, _iswprint_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l	Printable
ispunct, iswpunct, _ispunct_l, _iswpunct_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l	Punctuation
isspace, iswspace, _isspace_l, _iswspace_l, _ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l	White-space
isupper, iswupper, _ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	Uppercase
_isctype, iswctype, _isctype_l, _iswctype_l	Property specified by *desc* argument
isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l	Hexadecimal digit
_mbclen, mblen, _mblen_l	Return length of valid multibyte character; result depends on **LC_CTYPE** category setting of current locale

See also

Universal C runtime routines by category

C complex math support

5/15/2019 • 3 minutes to read • Edit Online

The Microsoft C Runtime library (CRT) provides complex math library functions, including all of those required by ISO C99. The compiler does not directly support a **complex** or **_Complex** keyword, therefore the Microsoft implementation uses structure types to represent complex numbers.

These functions are implemented to balance performance with correctness. Because producing the correctly rounded result may be prohibitively expensive, these functions are designed to efficiently produce a close approximation to the correctly rounded result. In most cases, the result produced is within +/-1 ulp of the correctly rounded result, though there may be cases where there is greater inaccuracy.

The complex math routines rely on the floating point math library functions for their implementation. These functions have different implementations for different CPU architectures. For example, the 32-bit x86 CRT may have a different implementation than the 64-bit x64 CRT. In addition, some of the functions may have multiple implementations for a given CPU architecture. The most efficient implementation is selected dynamically at run-time depending on the instruction sets supported by the CPU. For example, in the 32-bit x86 CRT, some functions have both an x87 implementation and an SSE2 implementation. When running on a CPU that supports SSE2, the faster SSE2 implementation is used. When running on a CPU that does not support SSE2, the slower x87 implementation is used. Because different implementations of the math library functions may use different CPU instructions and different algorithms to produce their results, the functions may produce different results across CPUs. In most cases, the results are within +/-1 ulp of the correctly rounded result, but the actual results may vary across CPUs.

Types used in complex math

The Microsoft implementation of the complex.h header defines these types as equivalents for the C99 standard native complex types:

STANDARD TYPE	MICROSOFT TYPE
float complex or **float _Complex**	**_Fcomplex**
double complex or **double _Complex**	**_Dcomplex**
long double complex or **long double _Complex**	**_Lcomplex**

The math.h header defines a separate type, **struct _complex**, used for the _cabs function. The **struct _complex** type is not used by the equivalent complex math functions cabs, cabsf, cabsl.

Complex constants and macros

I is defined as the **float** complex type **_Fcomplex** initialized by `{ 0.0f, 1.0f }`.

Trigonometric functions

FUNCTION	DESCRIPTION
cacos, cacosf, cacosl	Compute the complex arc cosine of a complex number

FUNCTION	DESCRIPTION
casin, casinf, casinl	Compute the complex arc sine of a complex number
catan, catanf, catanl	Compute the complex arc tangent of a complex number
ccos, ccosf, ccosl	Compute the complex cosine of a complex number
csin, csinf, csinl	Compute the complex sine of a complex number
ctan, ctanf, ctanl	Compute the complex tangent of a complex number

Hyperbolic functions

FUNCTION	DESCRIPTION
cacosh, cacoshf, cacoshl	Compute the complex arc hyperbolic cosine of a complex number
casinh, casinhf, casinhl	Compute the complex arc hyperbolic sine of a complex number
catanh, catanhf, catanhl	Compute the complex arc hyperbolic tangent of a complex number
ccosh, ccoshf, ccoshl	Compute the complex hyperbolic cosine of a complex number
csinh, csinhf, csinhl	Compute the complex hyperbolic sine of a complex number
ctanh, ctanhf, ctanhl	Compute the complex hyperbolic tangent of a complex number

Exponential and logarithmic functions

FUNCTION	DESCRIPTION
cexp, cexpf, cexpl	Compute the complex base-*e* exponential of a complex number
clog, clogf, clogl	Compute the complex natural (base-*e*) logarithm of a complex number
clog10, clog10f, clog10l	Compute the complex base-10 logarithm of a complex number

Power and absolute-value functions

FUNCTION	DESCRIPTION
cabs, cabsf, cabsl	Compute the complex absolute value (also called the norm, modulus, or magnitude) of a complex number

FUNCTION	DESCRIPTION
cpow, cpowf, cpowl	Compute the complex power function x^y
csqrt, csqrtf, csqrtl	Compute the complex square root of a complex number

Manipulation functions

FUNCTION	DESCRIPTION
_Cbuild, _FCbuild, _LCbuild	Construct a complex number from real and imaginary parts
carg, cargf, cargl	Compute the argument (also called the phase angle) of a complex number
cimag, cimagf, cimagl	Compute the imaginary part of a complex number
conj, conjf, conjl	Compute the complex conjugate of a complex number
cproj, cprojf, cprojl	Compute a projection of a complex number onto the Riemann sphere
creal, crealf, creall	Compute the real part of a complex number
norm, normf, norml	Compute the squared magnitude of a complex number

Operation functions

Because complex numbers are not a native type in the Microsoft compiler, the standard arithmetic operators are not defined on complex types. For convenience, these complex math library functions are provided to enable limited manipulation of complex numbers in user code:

FUNCTION	DESCRIPTION
_Cmulcc, _FCmulcc, _LCmulcc	Multiply two complex numbers
_Cmulcr, _FCmulcr, _LCmulcr	Multiply a complex and a floating-point number

See also

Universal C runtime routines by category

Data Alignment

3/11/2019 • 2 minutes to read • Edit Online

The following C run-time functions support data alignment.

Data-Alignment Routines

ROUTINE	USE
_aligned_free	Frees a block of memory that was allocated with _aligned_mallocor _aligned_offset_malloc.
_aligned_free_dbg	Frees a block of memory that was allocated with _aligned_malloc or _aligned_offset_malloc (debug only).
_aligned_malloc	Allocates memory on a specified alignment boundary.
_aligned_malloc_dbg	Allocates memory on a specified alignment boundary with additional space for a debugging header and overwrite buffers (debug version only).
_aligned_msize	Returns the size of a memory block allocated in the heap.
_aligned_msize_dbg	Returns the size of a memory block allocated in the heap (debug version only).
_aligned_offset_malloc	Allocates memory on a specified alignment boundary.
_aligned_offset_malloc_dbg	Allocates memory on a specified alignment boundary (debug version only).
_aligned_offset_realloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.
_aligned_offset_realloc_dbg	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc (debug version only).
_aligned_offset_recalloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.
_aligned_offset_recalloc_dbg	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0 (debug version only).
_aligned_realloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.
_aligned_realloc_dbg	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc (debug version only).

ROUTINE	USE
_aligned_recalloc	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.
_aligned_recalloc_dbg	Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0 (debug version only).

See also

Universal C runtime routines by category

Data Conversion

3/5/2019 • 2 minutes to read • Edit Online

These routines convert data from one form to another. Generally these routines execute faster than conversions you might write. Each routine that begins with a *to* prefix is implemented as a function and as a macro. See Choosing Between Functions and Macros for information about choosing an implementation.

Data-conversion routines

ROUTINE	USE
abs	Find absolute value of integer
atof, _atof_l	Convert string to **float**
atoi, _atoi_l	Convert string to **int**
_atoi64, _atoi64_l	Convert string to **__int64** or **long long**
atol, _atol_l	Convert string to **long**
c16rtomb, c32rtomb	Convert UTF-16 or UTF-32 character to equivalent multibyte character
_ecvt, _ecvt_s	Convert **double** to string of specified length
_fcvt, _fcvt_s	Convert **double** to string with specified number of digits following decimal point
_gcvt, _gcvt_s	Convert **double** number to string; store string in buffer
_itoa, _ltoa, _ultoa, _i64toa, _ui64toa, _itow, _ltow, ultow, _i64tow, _ui64tow, _itoa_s, _ltoa_s, _ultoa_s, _i64toa_s, _ui64toa_s, _itow_s, _ltow_s, _ultow_s, _i64tow_s, _ui64tow_s	Convert integer types to string
labs	Find absolute value of **long** integer
llabs	Find absolute value of **long long** integer
_mbbtombc, _mbbtombc_l	Convert 1-byte multibyte character to corresponding 2-byte multibyte character
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l	Convert Japan Industry Standard (JIS) character to Japan Microsoft (JMS) character
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l	Convert JMS character to JIS character
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l	Convert multibyte character to 1-byte hiragana code

ROUTINE	USE
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l	Convert multibyte character to 1-byte katakana code
_mbctombb, _mbctombb_l	Convert 2-byte multibyte character to corresponding 1-byte multibyte character
mbrtoc16, mbrtoc32	Convert multibyte character to equivalent UTF-16 or UTF-32 character
mbstowcs, _mbstowcs_l, mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters
mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character
strtod, _strtod_l, wcstod, _wcstod_l	Convert string to **double**
strtol, wcstol, _strtol_l, _wcstol_l	Convert string to **long** integer
strtoul, _strtoul_l, wcstoul, _wcstoul_l	Convert string to **unsigned long** integer
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form based on locale-specific information
toascii, __toascii	Convert character to ASCII code
tolower, _tolower, towlower, _tolower_l, _towlower_l, _mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Test character and convert to lowercase if currently uppercase
tolower, _tolower, towlower, _tolower_l, _towlower_l	Convert character to lowercase unconditionally
toupper, _toupper, towupper, _toupper_l, _towupper_l, _mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Test character and convert to uppercase if currently lowercase
toupper, _toupper, towupper, _toupper_l, _towupper_l	Convert character to uppercase unconditionally
wcstombs, _wcstombs_l, wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters
wctomb, _wctomb_l, wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character
_wtof, _wtof_l	Convert wide-character string to a **double**
_wtoi, _wtoi_l	Convert wide-character string to **int**
_wtoi64, _wtoi64_l	Convert wide-character string to **__int64** or **long long**
_wtol, _wtol_l	Convert wide-character string to **long**

See also

Universal C runtime routines by category

Debug routines

11/8/2018 • 4 minutes to read • Edit Online

The debug version of the C runtime library supplies many diagnostic services that make debugging programs easier and allow developers to:

- Step directly into run-time functions during debugging
- Resolve assertions, errors, and exceptions
- Trace heap allocations and prevent memory leaks
- Report debug messages to the user

Debug versions of the C runtime library routines

To use these routines, the _DEBUG flag must be defined. All of these routines do nothing in a retail build of an application. For more information on how to use the new debug routines, see CRT Debugging Techniques.

ROUTINE	USE
_ASSERT	Evaluate an expression and generates a debug report when the result is FALSE
_ASSERTE	Similar to **_ASSERT**, but includes the failed expression in the generated report
_CrtCheckMemory	Confirm the integrity of the memory blocks allocated on the debug heap
_CrtDbgBreak	Sets a break point.
_CrtDbgReport, _CrtDbgReportW	Generate a debug report with a user message and send the report to three possible destinations
_CrtDoForAllClientObjects	Call an application-supplied function for all **_CLIENT_BLOCK** types on the heap
_CrtDumpMemoryLeaks	Dump all of the memory blocks on the debug heap when a significant memory leak has occurred
_CrtIsMemoryBlock	Verify that a specified memory block is located within the local heap and that it has a valid debug heap block type identifier
_CrtIsValidHeapPointer	Verifies that a specified pointer is in the local heap
_CrtIsValidPointer	Verify that a specified memory range is valid for reading and writing
_CrtMemCheckpoint	Obtain the current state of the debug heap and store it in an application-supplied **_CrtMemState** structure

ROUTINE	USE
_CrtMemDifference	Compare two memory states for significant differences and return the results
_CrtMemDumpAllObjectsSince	Dump information about objects on the heap since a specified checkpoint was taken or from the start of program execution
_CrtMemDumpStatistics	Dump the debug header information for a specified memory state in a user-readable form
_CrtReportBlockType	Returns the block type/subtype associated with a given debug heap block pointer.
_CrtSetAllocHook	Install a client-defined allocation function by hooking it into the C run-time debug memory allocation process
_CrtSetBreakAlloc	Set a breakpoint on a specified object allocation order number
_CrtSetDbgFlag	Retrieve or modify the state of the **_crtDbgFlag** flag to control the allocation behavior of the debug heap manager
_CrtSetDumpClient	Install an application-defined function that is called every time a debug dump function is called to dump **_CLIENT_BLOCK** type memory blocks
_CrtSetReportFile	Identify the file or stream to be used as a destination for a specific report type by **_CrtDbgReport**
_CrtSetReportHook	Install a client-defined reporting function by hooking it into the C run-time debug reporting process
_CrtSetReportHook2, _CrtSetReportHookW2	Installs or uninstalls a client-defined reporting function by hooking it into the C run-time debug reporting process.
_CrtSetReportMode	Specify the general destination(s) for a specific report type generated by **_CrtDbgReport**
_RPT[0,1,2,3,4]	Track the application's progress by generating a debug report by calling **_CrtDbgReport** with a format string and a variable number of arguments. Provides no source file and line number information.
_RPTF[0,1,2,3,4]	Similar to the **_RPTn** macros, but provides the source file name and line number where the report request originated
_calloc_dbg	Allocate a specified number of memory blocks on the heap with additional space for a debugging header and overwrite buffers
_expand_dbg	Resize a specified block of memory on the heap by expanding or contracting the block
_free_dbg	Free a block of memory on the heap

ROUTINE	USE
_fullpath_dbg, _wfullpath_dbg	Create an absolute or full path name for the specified relative path name, using _malloc_dbg to allocate memory.
_getcwd_dbg, _wgetcwd_dbg	Get the current working directory, using _malloc_dbg to allocate memory.
_malloc_dbg	Allocate a block of memory on the heap with additional space for a debugging header and overwrite buffers
_msize_dbg	Calculate the size of a block of memory on the heap
_realloc_dbg	Reallocate a specified block of memory on the heap by moving and/or resizing the block
_strdup_dbg, _wcsdup_dbg	Duplicates a string, using _malloc_dbg to allocate memory.
_tempnam_dbg, _wtempnam_dbg	Generate names you can use to create temporary files, using _malloc_dbg to allocate memory.

C runtime routines that are not available in source code form

The debugger can be used to step through the source code for most of the C runtime routines during the debugging process. However, Microsoft considers some technology to be proprietary and, therefore, does not provide the source code for a subset of these routines. Most of these routines belong to either the exception handling or floating-point processing groups, but a few others are included as well. The following table lists these routines.

acos	acosh	asin
asinh	atan, atan2	atanh
Bessel functions	_cabs	ceil
_chgsign	_clear87, _clearfp	_control87, _controlfp
copysign	cos	cosh
Exp	fabs	_finite
floor	fmod	_fpclass
_fpieee_flt	_fpreset	frexp
_hypot	_isnan	ldexp
log	_logb	log10
longjmp	_matherr	modf

_nextafter	pow	printf_s
printf	_scalb	scanf_s
scanf	setjmp	sin
sinh	sqrt	_status87, _statusfp
tan	tanh	

Although source code is available for most of the **printf** and **scanf** routines, they make an internal call to another routine for which source code is not provided.

Routines that behave differently in a debug build of an application

Some C run-time functions and C++ operators behave differently when called from a debug build of an application. (Note that a debug build of an application can be done by either defining the `_DEBUG` flag or by linking with a debug version of the C run-time library.) The behavioral differences usually consist of extra features or information provided by the routine to support the debugging process. The following table lists these routines.

C abort routine	C++ delete operator
C assert routine	C++ new operator

See also

Universal C runtime routines by category
Run-Time Error Checking

Directory Control

3/11/2019 • 2 minutes to read • Edit Online

These routines access, modify, and obtain information about the directory structure.

Directory-Control Routines

ROUTINE	USE
_chdir, _wchdir	Change current working directory
_chdrive	Change current drive
_getcwd, _wgetcwd	Get current working directory for default drive
_getdcwd, _wgetdcwd	Get current working directory for specified drive
_getdiskfree	Populates a **_diskfree_t** structure with information about a disk drive.
_getdrive	Get current (default) drive
_getdrives	Returns a bitmask representing the currently available disk drives.
_mkdir, _wmkdir	Make new directory
_rmdir, _wrmdir	Remove directory
_searchenv, _wsearchenv, _searchenv_s, _wsearchenv_s	Search for given file on specified paths

See also

Universal C runtime routines by category
File Handling
System Calls

Error handling (CRT)

10/31/2018 • 2 minutes to read • Edit Online

Use these routines to handle program errors.

Error-handling routines

ROUTINE	USE
assert macro	Test for programming logic errors; available in both the release and debug versions of the run-time library.
_ASSERT, _ASSERTE macros	Similar to **assert**, but only available in the debug versions of the run-time library.
clearerr	Reset error indicator. Calling **rewind** or closing a stream also resets the error indicator.
_eof	Check for end of file in low-level I/O.
feof	Test for end of file. End of file is also indicated when **_read** returns 0.
ferror	Test for stream I/O errors.
_RPT, _RPTF macros	Generate a report similar to **printf**, but only available in the debug versions of the run-time library.
_set_error_mode	Modifies **__error_mode** to determine a non-default location where the C run time writes an error message for an error that will possibly end the program.
_set_purecall_handler	Sets the handler for a pure virtual function call.

See also

- Universal C runtime routines by category

Exception Handling Routines

3/11/2019 • 2 minutes to read • Edit Online

Use the C++ exception-handling functions to recover from unexpected events during program execution.

Exception-Handling Functions

FUNCTION	USE
_set_se_translator	Handle Win32 exceptions (C structured exceptions) as C++ typed exceptions
set_terminate	Install your own termination routine to be called by **terminate**
set_unexpected	Install your own termination function to be called by **unexpected**
terminate	Called automatically under certain circumstances after exception is thrown. The **terminate** function calls **abort** or a function you specify using **set_terminate**
unexpected	Calls **terminate** or a function you specify using **set_unexpected**. The **unexpected** function is not used in current Microsoft C++ exception-handling implementation

See also

Universal C runtime routines by category

File Handling

3/11/2019 • 2 minutes to read • Edit Online

Use these routines to create, delete, and manipulate files and to set and check file-access permissions.

The C run-time libraries have a 512 limit for the number of files that can be open at any one time. Attempting to open more than the maximum number of file descriptors or file streams causes program failure. Use _setmaxstdio to change this number.

File-Handling Routines (File Descriptor)

These routines operate on files designated by a file descriptor.

ROUTINE	USE
_chsize,_chsize_s	Change file size
_filelength, _filelengthi64	Get file length
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32	Get file-status information on descriptor
_get_osfhandle	Return operating-system file handle associated with existing C run-time file descriptor
_isatty	Check for character device
_locking	Lock areas of file
_open_osfhandle	Associate C run-time file descriptor with existing operating-system file handle
_setmode	Set file-translation mode

File-Handling Routines (Path or Filename)

These routines operate on files specified by a path or filename.

ROUTINE	USE
_access, _waccess, _access_s, _waccess_s	Check file-permission setting
_chmod, _wchmod	Change file-permission setting
_fullpath, _wfullpath	Expand a relative path to its absolute path name
_makepath, _wmakepath, _makepath_s, _wmakepath_s	Merge path components into single, full path
_mktemp, _wmktemp, _mktemp_s, _wmktemp_s	Create unique filename

ROUTINE	USE
remove, _wremove	Delete file
rename, _wrename	Rename file
_splitpath, _wsplitpath, _splitpath_s, _wsplitpath_s	Parse path into components
_stat, _stat64, _stati64, _wstat, _wstat64, _wstati64	Get file-status information on named file
_umask, _umask_s	Set default permission mask for new files created by program
_unlink, _wunlink	Delete file

File-Handling Routines (Open File)

These routines open files.

ROUTINE	USE
fopen, _wfopen, fopen_s, _wfopen_s	Opens a file and returns a pointer to the open file.
_fsopen, _wfsopen	Open a stream with file sharing and returns a pointer to the open file.
_open, _wopen	Opens a file and returns a file descriptor to the opened file.
_sopen, _wsopen, _sopen_s, _wsopen_s	Open a file with file sharing and returns a file descriptor to the open file.
_pipe	Creates a pipe for reading and writing.
freopen, _wfreopen, freopen_s, _wfreopen_s	Reassign a file pointer.

These routines provide a way to change the representation of the file between a `FILE` structure, a file descriptor, and a Win32 file handle.

ROUTINE	USE
_fdopen, _wfdopen	Associates a stream with a file that was previously opened for low-level I/O and returns a pointer to the open stream.
_fileno	Gets the file descriptor associated with a stream.
_get_osfhandle	Return operating-system file handle associated with existing C run-time file descriptor
_open_osfhandle	Associates C run-time file descriptor with an existing operating-system file handle.

The following Win32 functions also open files and pipes:

- CreateFile

- CreatePipe
- CreateNamedPipe

See also

Universal C runtime routines by category
Directory Control
System Calls

Math and floating-point support

2/4/2019 • 6 minutes to read • Edit Online

The Universal C Runtime library (UCRT) provides many integral and floating-point math library functions, including all of those required by ISO C99. The floating-point functions are implemented to balance performance with correctness. Because producing the correctly rounded result may be prohibitively expensive, these functions are designed to efficiently produce a close approximation to the correctly rounded result. In most cases, the result produced is within +/-1 ulp of the correctly rounded result, though there may be cases where there is greater inaccuracy.

Many of the floating point math library functions have different implementations for different CPU architectures. For example, the 32-bit x86 CRT may have a different implementation than the 64-bit x64 CRT. In addition, some of the functions may have multiple implementations for a given CPU architecture. The most efficient implementation is selected dynamically at run-time depending on the instruction sets supported by the CPU. For example, in the 32-bit x86 CRT, some functions have both an x87 implementation and an SSE2 implementation. When running on a CPU that supports SSE2, the faster SSE2 implementation is used. When running on a CPU that does not support SSE2, the slower x87 implementation is used. Because different implementations of the math library functions may use different CPU instructions and different algorithms to produce their results, the functions may produce different results across CPUs. In most cases, the results are within +/-1 ulp of the correctly rounded result, but the actual results may vary across CPUs.

Previous 16-bit versions of Microsoft C/C++ and Microsoft Visual C++ supported the **long double** type as an 80-bit precision floating-point data type. In later versions of Visual C++, the **long double** data type is a 64-bit precision floating-point data type identical to the **double** type. The compiler treats **long double** and **double** as distinct types, but the **long double** functions are identical to their **double** counterparts. The CRT provides **long double** versions of the math functions for ISO C99 source code compatibility, but note that the binary representation may differ from other compilers.

Supported math and floating-point routines

ROUTINE	USE
abs, labs, llabs, _abs64	Computes the absolute value of an integer type
acos, acosf, acosl	Computes the arc cosine
acosh, acoshf, acoshl	Computes the hyperbolic arc cosine
asin, asinf, asinl	Computes the arc sine
asinh, asinhf, asinhl	Computes the hyperbolic arc sine
atan, atanf, atanl, atan2, atan2f, atan2l	Computes the arc tangent
atanh, atanhf, atanhl	Computes the hyperbolic arc tangent
_atodbl, _atodbl_l	Converts a locale-specific string to a **double**

ROUTINE	USE
atof, _atof_l	Converts a string to a **double**
_atoflt, _atoflt_l, _atoldbl, _atoldbl_l	Converts a locale-specific string to a **float** or **long double**
cbrt, cbrtf, cbrtl	Computes the cube root
ceil, ceilf, ceill	Computes the ceiling
_chgsign, _chgsignf, _chgsignl	Computes the additive inverse
_clear87, _clearfp	Gets and clears the floating-point status register
_control87, __control87_2, _controlfp	Gets and sets the floating-point control word
_controlfp_s	Secure version of **_controlfp**
copysign, copysignf, copysignl, _copysign, _copysignf, _copysignl	Returns a value that has the magnitude of one argument and the sign of another
cos, cosf, cosl	Computes the sine
cosh, coshf, coshl	Computes the hyperbolic sine
div, ldiv, lldiv	Computes the quotient and the remainder of two integer values
_ecvt, ecvt	Converts a **double** to a string
_ecvt_s	Secure version of **_ecvt**
erf, erff, erfl	Computes the error function
erfc, erfcf, erfcl	Computes the complementary error function
exp, expf, expl	Computes the exponential e^x
exp2, exp2f, exp2l	Computes the exponential 2^x
expm1, expm1f, expm1l	Computes e^x-1
fabs, fabsf, fabsl	Computes the absolute value of a floating-point type
_fcvt, fcvt	Converts a floating-point number to a string
_fcvt_s	Secure version of **_fcvt**
fdim, fdimf, fdiml	Determines the positive difference between two values
feclearexcept	Clears specified floating-point exceptions

ROUTINE	USE
fegetenv	Stores the current floating-point environment
fegetexceptflag	Gets the specified floating-point exception status
fegetround	Gets the floating-point rounding mode
feholdexcept	Sets non-stop floating-point exception mode
feraiseexcept	Raises the specified floating-point exceptions
fesetenv	Sets the current floating-point environment
fesetexceptflag	Sets the specified floating-point status flags
fesetround	Sets the specified floating-point rounding mode
fetestexcept	Determines which floating-point exception status flags are set
feupdateenv	Restores a floating-point environment then raises previous exceptions
floor, floorf, floorl	Computes the floor
fma, fmaf, fmal	Computes a fused multiply-add
fmax, fmaxf, fmaxl	Computes the maximum of the arguments
fmin, fminf, fminl	Computes the minumum of the arguments
fmod, fmodf, fmodl	Computes the floating-point remainder
_fpclass, _fpclassf	Returns the classification of a floating-point value
fpclassify	Returns the classification of a floating-point value
_fpieee_flt	Sets a handler for floating-point exceptions
_fpreset	Resets the floating-point environment
frexp, frexpf, frexpl	Gets the mantissa and exponent of a floating-point number
_gcvt, gcvt	Converts a floating-point number to a string
_gcvt_s	Secure version of **_gcvt**
_get_FMA3_enable, _set_FMA3_enable	Gets or sets a flag for use of FMA3 instructions on x64
hypot, hypotf, hypotl, _hypot, _hypotf, _hypotl	Computes the hypotenuse

ROUTINE	USE
ilogb, ilogbf, ilogbl	Computes the integer base-2 exponent
imaxabs	Computes the absolute value of an integer type
imaxdiv	Computes the quotient and the remainder of two integer values
isfinite, _finite, _finitef	Determines whether a value is finite
isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered	Compare the order of two floating point values
isinf	Determines whether a floating-point value is infinite
isnan, _isnan, _isnanf	Tests a floating-point value for NaN
isnormal	Tests whether a floating-point value is both finite and not subnormal
_j0, _j1, _jn	Computes the Bessel function
ldexp, ldexpf, ldexpl	Computes $x*2^n$
lgamma, lgammaf, lgammal	Computes the natural logarithm of the absolute value of the gamma function
llrint, llrintf, llrintl	Rounds a floating-point value to the nearest **long long** value
llround, llroundf, llroundl	Rounds a floating-point value to the nearest **long long** value
log, logf, logl, log10, log10f, log10l	Computes the natural or base-10 logarithm
log1p, log1pf, log1pl	Computes the natural logarithm of 1+x
log2, log2f, log2l	Computes the base-2 logarithm
logb, logbf, logbl, _logb, _logbf	Returns the exponent of a floating-point value
lrint, lrintf, lrintl	Rounds a floating-point value to the nearest **long** value
_lrotl, _lrotr	Rotates an integer value left or right
lround, lroundf, lroundl	Rounds a floating-point value to the nearest **long** value
_matherr	The default math error handler
__max	Macro that returns the larger of two values

ROUTINE	USE
__min	Macro that returns the smaller of two values
modf, modff, modfl	Splits a floating-point value into fractional and integer parts
nan, nanf, nanl	Returns a quiet NaN value
nearbyint, nearbyintf, nearbyintl	Returns the rounded value
nextafter, nextafterf, nextafterl, _nextafter, _nextafterf	Returns the next representable floating-point value
nexttoward, nexttowardf, nexttowardl	Returns the next representable floating-point value
pow, powf, powl	Returns the value of x^y
remainder, remainderf, remainderl	Computes the remainder of the quotient of two floating-point values
remquo, remquof, remquol	Computes the remainder of two integer values
rint, rintf, rintl	Rounds a floating-point value
_rotl, _rotl64, _rotr, _rotr64	Rotates bits in integer types
round, roundf, roundl	Rounds a floating-point value
_scalb, _scalbf	Scales argument by a power of 2
scalbn, scalbnf, scalbnl, scalbln, scalblnf, scalblnl	Multiplies a floating-point number by an integral power of **FLT_RADIX**
_set_controlfp	Sets the floating-point control word
_set_SSE2_enable	Enables or disables SSE2 instructions
signbit	Tests the sign bit of a floating-point value
sin, sinf, sinl	Computes the sine
sinh, sinhf, sinhl	Computes the hyperbolic sine
sqrt, sqrtf, sqrtl	Computes the square root
_status87, _statusfp, _statusfp2	Gets the floating-point status word
strtof, _strtof_l	Converts a string to a **float**
strtold, _strtold_l	Converts a string to a **long double**
tan, tanf, tanl	Computes the tangent

ROUTINE	USE
tanh, tanhf, tanhl	Computes the hyperbolic tangent
tgamma, tgammaf, tgammal	Computes the gamma function
trunc, truncf, truncl	Truncates the fractional part
_wtof, _wtof_l	Converts a wide string to a **double**
_y0, _y1, _yn	Computes the Bessel function

See also

Universal C runtime routines by category
Floating-point primitives

Input and Output

3/11/2019 • 2 minutes to read • Edit Online

The I/O functions read and write data to and from files and devices. File I/O operations take place in text mode or binary mode. The Microsoft run-time library has three types of I/O functions:

- Stream I/O functions treat data as a stream of individual characters.
- Low-level I/O functions invoke the operating system directly for lower-level operation than that provided by stream I/O.
- Console and port I/O functions read or write directly to a console (keyboard and screen) or an I/O port (such as a printer port).

> **NOTE**
> Because stream functions are buffered and low-level functions are not, these two types of functions are generally incompatible. For processing a particular file, use either stream or low-level functions exclusively.

See also

Universal C runtime routines by category

Text and Binary Mode File I/O

3/11/2019 • 2 minutes to read • Edit Online

File I/O operations take place in one of two translation modes, *text* or *binary*, depending on the mode in which the file is opened. Data files are usually processed in text mode. To control the file translation mode, one can:

- Retain the current default setting and specify the alternative mode only when you open selected files.
- Use the function _set_fmode to change the default mode for newly opened files. Use _get_fmode to find the current default mode. The initial default setting is text mode (**_O_TEXT**).
- Change the default translation mode directly by setting the global variable _fmode in your program. The function **_set_fmode** sets the value of this variable, but it can also be set directly.

When you call a file-open function such as _open, fopen, fopen_s, freopen, freopen_s, _fsopen or _sopen_s, you can override the current default setting of **_fmode** by specifying the appropriate argument to the function _set_fmode. The **stdin**, **stdout**, and **stderr** streams always open in text mode by default; you can also override this default when opening any of these files. Use _setmode to change the translation mode using the file descriptor after the file is open.

See also

Input and Output
Universal C runtime routines by category

Unicode Stream I/O in Text and Binary Modes

3/11/2019 • 2 minutes to read • Edit Online

When a Unicode stream I/O routine (such as **fwprintf**, **fwscanf**, **fgetwc**, **fputwc**, **fgetws**, or **fputws**) operates on a file that is open in text mode (the default), two kinds of character conversions take place:

- Unicode-to-MBCS or MBCS-to-Unicode conversion. When a Unicode stream-I/O function operates in text mode, the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).
- Carriage return - linefeed (CR-LF) translation. This translation occurs before the MBCS - Unicode conversion (for Unicode stream input functions) and after the Unicode - MBCS conversion (for Unicode stream output functions). During input, each carriage return - linefeed combination is translated into a single linefeed character. During output, each linefeed character is translated into a carriage return - linefeed combination.

However, when a Unicode stream-I/O function operates in binary mode, the file is assumed to be Unicode, and no CR-LF translation or character conversion occurs during input or output. Use the _setmode(_fileno(stdin), _O_BINARY); instruction in order to correctly use wcin on a UNICODE text file.

See also

Universal C runtime routines by category
Input and Output

Stream I/O

3/11/2019 • 4 minutes to read • Edit Online

These functions process data in different sizes and formats, from single characters to large data structures. They also provide buffering, which can improve performance. The default size of a stream buffer is 4K. These routines affect only buffers created by the run-time library routines, and have no effect on buffers created by the operating system.

Stream I/O Routines

ROUTINE	USE
clearerr, clearerr_s	Clear error indicator for stream
fclose	Close stream
_fcloseall	Close all open streams except **stdin**, **stdout**, and **stderr**
_fdopen, wfdopen	Associate stream with file descriptor of open file
feof	Test for end of file on stream
ferror	Test for error on stream
fflush	Flush stream to buffer or storage device
fgetc, fgetwc	Read character from stream (function versions of **getc** and **getwc**)
_fgetchar, _fgetwchar	Read character from **stdin** (function versions of **getchar** and **getwchar**)
fgetpos	Get position indicator of stream
fgets, fgetws	Read string from stream
_fileno	Get file descriptor associated with stream
_flushall	Flush all streams to buffer or storage device
fopen, _wfopen, fopen_s, _wfopen_s	Open stream
fprintf, _fprintf_l, fwprintf, _fwprintf_l, fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l	Write formatted data to stream
fputc, fputwc	Write a character to a stream (function versions of **putc** and **putwc**)

ROUTINE	USE
_fputchar, _fputwchar	Write character to **stdout** (function versions of **putchar** and **putwchar**)
fputs, fputws	Write string to stream
fread	Read unformatted data from stream
freopen, _wfreopen, freopen_s, _wfreopen_s	Reassign **FILE** stream pointer to new file or device
fscanf, fwscanf, fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l	Read formatted data from stream
fseek, _fseeki64	Move file position to given location
fsetpos	Set position indicator of stream
_fsopen, _wfsopen	Open stream with file sharing
ftell, _ftelli64	Get current file position
fwrite	Write unformatted data items to stream
getc, getwc	Read character from stream (macro versions of **fgetc** and **fgetwc**)
getchar, getwchar	Read character from **stdin** (macro versions of **fgetchar** and **fgetwchar**)
_getmaxstdio	Returns the number of simultaneously open files permitted at the stream I/O level.
gets_s, _getws_s	Read line from **stdin**
_getw	Read binary **int** from stream
printf, _printf_l, wprintf, _wprintf_l,printf_s, _printf_s_l, wprintf_s, _wprintf_s_l	Write formatted data to **stdout**
putc, putwc	Write character to a stream (macro versions of **fputc** and **fputwc**)
putchar, putwchar	Write character to **stdout** (macro versions of **fputchar** and **fputwchar**)
puts, _putws	Write line to stream
_putw	Write binary **int** to stream
rewind	Move file position to beginning of stream
_rmtmp	Remove temporary files created by **tmpfile**

ROUTINE	USE
scanf, _scanf_l, wscanf, _wscanf_l,scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l	Read formatted data from **stdin**
setbuf	Control stream buffering
_setmaxstdio	Set a maximum for the number of simultaneously open files at the stream I/O level.
setvbuf	Control stream buffering and buffer size
_snprintf, _snwprintf, _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l	Write formatted data of specified length to string
_snscanf, _snwscanf, _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from the standard input stream.
sprintf, swprintf, sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l	Write formatted data to string
sscanf, swscanf, sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data from string
_tempnam, _wtempnam	Generate temporary filename in given directory
tmpfile, tmpfile_s	Create temporary file
tmpnam, _wtmpnam, tmpnam_s, _wtmpnam_s	Generate temporary filename
ungetc, ungetwc	Push character back onto stream
_vcprintf, _vcwprintf, _vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l	Write formatted data to the console.
vfprintf, vfwprintf, vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l	Write formatted data to stream
vprintf, vwprintf, vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	Write formatted data to **stdout**
_vsnprintf, _vsnwprintf, vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted data of specified length to buffer
vsprintf, vswprintf, vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l	Write formatted data to buffer

When a program begins execution, the startup code automatically opens several streams: standard input (pointed to by **stdin**), standard output (pointed to by **stdout**), and standard error (pointed to by **stderr**). These streams are directed to the console (keyboard and screen) by default. Use **freopen** to redirect **stdin**, **stdout**, or **stderr** to a disk file or a device.

Files opened using the stream routines are buffered by default. The **stdout** and **stderr** functions are flushed whenever they are full or, if you are writing to a character device, after each library call. If a program terminates abnormally, output buffers may not be flushed, resulting in loss of data. Use

fflush or **_flushall** to ensure that the buffer associated with a specified file or all open buffers are flushed to the operating system, which can cache data before writing it to disk. The commit-to-disk feature ensures that the flushed buffer contents are not lost in the event of a system failure.

There are two ways to commit buffer contents to disk:

- Link with the file COMMODE.OBJ to set a global commit flag. The default setting of the global flag is **n**, for "no-commit."
- Set the mode flag to **c** with **fopen** or **_fdopen**.

Any file specifically opened with either the **c** or the **n** flag behaves according to the flag, regardless of the state of the global commit/no-commit flag.

If your program does not explicitly close a stream, the stream is automatically closed when the program terminates. However, you should close a stream when your program finishes with it, as the number of streams that can be open at one time is limited. See _setmaxstdio for information on this limit.

Input can follow output directly only with an intervening call to **fflush** or to a file-positioning function (**fseek**, **fsetpos**, or **rewind**). Output can follow input without an intervening call to a file-positioning function if the input operation encounters the end of the file.

See also

Input and Output
Universal C runtime routines by category

Low-Level I/O

3/11/2019 • 2 minutes to read • Edit Online

These functions invoke the operating system directly for lower-level operation than that provided by stream I/O. Low-level input and output calls do not buffer or format data.

Low-level routines can access the standard streams opened at program startup using the following predefined file descriptors.

STREAM	FILE DESCRIPTOR
stdin	0
stdout	1
stderr	2

Low-level I/O routines set the errno global variable when an error occurs. You must include STDIO.H when you use low-level functions only if your program requires a constant that is defined in STDIO.H, such as the end-of-file indicator (**EOF**).

Low-Level I/O Functions

FUNCTION	USE
_close	Close file
_commit	Flush file to disk
_creat, _wcreat	Create file
_dup	Return next available file descriptor for given file
_dup2	Create second descriptor for given file
_eof	Test for end of file
_lseek, _lseeki64	Reposition file pointer to given location
_open, _wopen	Open file
_read	Read data from file
_sopen, _wsopen, _sopen_s, _wsopen_s	Open file for file sharing
_tell, _telli64	Get current file-pointer position
_umask, _umask_s	Set file-permission mask

FUNCTION	USE
_write	Write data to file

_dup and **_dup2** are typically used to associate the predefined file descriptors with different files.

See also

Input and Output
Universal C runtime routines by category
System Calls

Console and Port I/O

3/11/2019 • 2 minutes to read • Edit Online

These routines read and write on your console or on the specified port. The console I/O routines are not compatible with stream I/O or low-level I/O library routines. The console or port does not have to be opened or closed before I/O is performed, so there are no open or close routines in this category. In the Windows operating systems, the output from these functions is always directed to the console and cannot be redirected.

Console and Port I/O Routines

ROUTINE	USE
_cgets, _cgetws, _cgets_s, _cgetws_s	Read string from console
_cprintf, _cwprintf, _cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l	Write formatted data to console
_cputs	Write string to console
_cscanf, _cwscanf, _cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l	Read formatted data from console
_getch, _getwch	Read character from console
_getche, _getwche	Read character from console and echo it
_inp	Read one byte from specified I/O port
_inpd	Read double word from specified I/O port
_inpw	Read 2-byte word from specified I/O port
_kbhit	Check for keystroke at console; use before attempting to read from console
_outp	Write one byte to specified I/O port
_outpd	Write double word to specified I/O port
_outpw	Write word to specified I/O port
_putch, _putwch	Write character to console
_ungetch, _ungetwch	"Unget" last character read from console so it becomes next character read

See also

Input and Output

Universal C runtime routines by category

_nolock Functions

3/11/2019 • 2 minutes to read • Edit Online

These are functions that do not perform any locking. They are provided for users requiring maximum performance. For more information, see Multithreaded Libraries Performance.

Use _nolock functions only if your program is truly single-threaded or if it does its own locking.

No lock routines

_fclose_nolock

_fflush_nolock

_fgetc_nolock, _fgetwc_nolock

_fread_nolock

_fseek_nolock, _fseeki64_nolock

_ftell_nolock, _ftelli64_nolock

_fwrite_nolock

_getc_nolock, _getwc_nolock

_getch_nolock, _getwch_nolock

_getchar_nolock, _getwchar_nolock

_getche_nolock, _getwche_nolock

_getdcwd_nolock, _wgetdcwd_nolock

_putc_nolock, _putwc_nolock

_putch_nolock, _putwch_nolock

_putchar_nolock, _putwchar_nolock

_ungetc_nolock, _ungetwc_nolock

_ungetch_nolock, _ungetwch_nolock

See also

Input and Output
Universal C runtime routines by category

Internationalization

3/11/2019 • 2 minutes to read • Edit Online

The Microsoft run-time library provides many routines that are useful for creating different versions of a program for international markets. This includes locale-related routines, wide-character routines, multibyte-character routines, and generic-text routines. For convenience, most locale-related routines are also categorized in this reference according to the operations they perform. In this section and in the alphabetic reference, multibyte-character routines and wide-character routines are described with single-byte-character counterparts, where they exist.

Also included are the ISO646 operator alternatives.

See also

Universal C runtime routines by category

Locale

3/11/2019 • 3 minutes to read • Edit Online

Locale refers to country/region and language settings that you can use to customize your program. Some locale-dependent categories include the display formats for dates and monetary values. For more information, see Locale Categories.

Use the setlocale function to change or query some or all of the current program or thread locale information while using functions without the **_l** suffix. The functions with the **_l** suffix will use the locale parameter passed in for their locale information during the execution of that specific function only. To create a locale for use with a function with a **_l** suffix, use _create_locale. To free this locale, use _free_locale. To get the current locale, use _get_current_locale.

Use _configthreadlocale to control whether each thread has its own locale, or all threads in a program share the same locale. For more information, see Locales and Code Pages.

More secure versions of the functions in the following table are available, indicated by the **_s** ("secure") suffix. For more information, see Security Features in the CRT.

Locale-Dependent Routines

ROUTINE	USE	SETLOCALE CATEGORY SETTING DEPENDENCE
atof, _atof_l, _wtof, _wtof_l	Convert character to floating-point value	**LC_NUMERIC**
atoi, _atoi_l, _wtoi, _wtoi_l	Convert character to integer value	**LC_NUMERIC**
_atoi64, _atoi64_l, _wtoi64, _wtoi64_l	Convert character to 64-bit integer value	**LC_NUMERIC**
atol, _atol_l, _wtol, _wtol_l	Convert character to long value	**LC_NUMERIC**
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l	Convert character to double-long value	**LC_NUMERIC**
is Routines	Test given integer for particular condition.	**LC_CTYPE**
isleadbyte, _isleadbyte_l	Test for lead byte	**LC_CTYPE**
localeconv	Read appropriate values for formatting numeric quantities	`LC_MONETARY, LC_NUMERIC`

ROUTINE	USE	SETLOCALE CATEGORY SETTING DEPENDENCE
MB_CUR_MAX	Maximum length in bytes of any multibyte character in current locale (macro defined in STDLIB.H)	**LC_CTYPE**
_mbccpy, _mbccpy_l,_mbccpy_s, _mbccpy_s_l	Copy one multibyte character	**LC_CTYPE**
_mbclen, mblen, _mblen_l	Validate and return number of bytes in multibyte character	**LC_CTYPE**
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l	For multibyte-character strings: validate each character in string; return string length	**LC_CTYPE**
mbstowcs, _mbstowcs_l,mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters	**LC_CTYPE**
mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character	**LC_CTYPE**
printf functions	Write formatted output	**LC_NUMERIC** (determines radix character output)
scanf functions	Read formatted input	**LC_NUMERIC** (determines radix character recognition)
setlocale, _wsetlocale	Select locale for program	Not applicable
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l	Compare characters of two strings	**LC_COLLATE**
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l	Compare two strings without regard to case	**LC_CTYPE**
_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l	Compare characters of two strings (case insensitive)	**LC_COLLATE**
_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l	Compare first **n** characters of two strings	**LC_COLLATE**
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l	Compare characters of two strings without regard to case.	**LC_CTYPE**

ROUTINE	USE	SETLOCALE CATEGORY SETTING DEPENDENCE
_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare first **n** characters of two strings (case insensitive)	**LC_COLLATE**
strftime, wcsftime, _strftime_l, _wcsftime_l	Format date and time value according to supplied **format** argument	**LC_TIME**
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l,_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l	Convert, in place, each uppercase letter in given string to lowercase	**LC_CTYPE**
strtod, _strtod_l, wcstod, _wcstod_l	Convert character string to **double** value	**LC_NUMERIC** (determines radix character recognition)
strtol, wcstol, _strtol_l, _wcstol_l	Convert character string to **long** value	**LC_NUMERIC** (determines radix character recognition)
strtoul, _strtoul_l, wcstoul, _wcstoul_l	Convert character string to unsigned long value	**LC_NUMERIC** (determines radix character recognition)
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr,_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert, in place, each lowercase letter in string to uppercase	**LC_CTYPE**
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form according to locale	**LC_COLLATE**
tolower, _tolower, towlower, _tolower_l, _towlower_l,_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Convert given character to corresponding lowercase character	**LC_CTYPE**
toupper, _toupper, towupper, _toupper_l, _towupper_l,_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l	Convert given character to corresponding uppercase letter	**LC_CTYPE**
wcstombs, _wcstombs_l,wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters	**LC_CTYPE**

ROUTINE	USE	SETLOCALE CATEGORY SETTING DEPENDENCE
wctomb, _wctomb_l,wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character	**LC_CTYPE**

> **NOTE**
>
> For multibyte routines, the multibyte code page must be equivalent to the locale set with setlocale. _setmbcp, with an argument of **_MB_CP_LOCALE** makes the multibyte code page the same as the **setlocale** code page.

See also

Internationalization
Universal C runtime routines by category

Code Pages

3/11/2019 • 2 minutes to read • Edit Online

A *code page* is a character set, which can include numbers, punctuation marks, and other glyphs. Different languages and locales may use different code pages. For example, ANSI code page 1252 is used for English and most European languages; OEM code page 932 is used for Japanese Kanji.

A code page can be represented in a table as a mapping of characters to single-byte values or multibyte values. Many code pages share the ASCII character set for characters in the range 0x00 - 0x7F.

The Microsoft run-time library uses the following types of code pages:

- System-default ANSI code page. By default, at startup the run-time system automatically sets the multibyte code page to the system-default ANSI code page, which is obtained from the operating system. The call:

  ```
  setlocale ( LC_ALL, "" );
  ```

  also sets the locale to the system-default ANSI code page.

- Locale code page. The behavior of a number of run-time routines is dependent on the current locale setting, which includes the locale code page. (For more information, see Locale-Dependent Routines.) By default, all locale-dependent routines in the Microsoft run-time library use the code page that corresponds to the "C" locale. At run-time you can change or query the locale code page in use with a call to setlocale.

- Multibyte code page. The behavior of most of the multibyte-character routines in the run-time library depends on the current multibyte code page setting. By default, these routines use the system-default ANSI code page. At run-time you can query and change the multibyte code page with _getmbcp and _setmbcp, respectively.

- The "C" locale is defined by ANSI to correspond to the locale in which C programs have traditionally executed. The code page for the "C" locale ("C" code page) corresponds to the ASCII character set. For example, in the "C" locale, **islower** returns true for the values 0x61 - 0x7A only. In another locale, **islower** may return true for these as well as other values, as defined by that locale.

See also

Internationalization
Universal C runtime routines by category

Interpretation of Multibyte-Character Sequences

3/11/2019 • 2 minutes to read • Edit Online

Most multibyte-character routines in the Microsoft run-time library recognize multibyte-character sequences relating to a multibyte code page. The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

Locale-Dependent Multibyte Routines

ROUTINE	USE
_mbclen, mblen, _mblen_l	Validate and return number of bytes in multibyte character
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l	For multibyte character strings: validate each character in string; return string length. For wide character strings: return string length.
mbstowcs, _mbstowcs_l, mbstowcs_s, _mbstowcs_s_l	Convert sequence of multibyte characters to corresponding sequence of wide characters
mbtowc, _mbtowc_l	Convert multibyte character to corresponding wide character
wcstombs, _wcstombs_l, wcstombs_s, _wcstombs_s_l	Convert sequence of wide characters to corresponding sequence of multibyte characters
wctomb, _wctomb_l, wctomb_s, _wctomb_s_l	Convert wide character to corresponding multibyte character
mbrtoc16, mbrtoc32	Convert multibyte character to equivalent UTF-16 or UTF-32 character
c16rtomb, c32rtomb	Convert UTF-16 or UTF-32 character to equivalent multibyte character

See also

Internationalization
Universal C runtime routines by category

ISO646 Operators

3/11/2019 • 2 minutes to read • Edit Online

Provides readable alternatives to certain operators or punctuators. The standard header <iso646.h> is available even in a freestanding implementation.

Macros

and	An alternative to the && operator.
and_eq	An alternative to the &= operator.
bitand	An alternative to the & operator.
bitor	An alternative to the | operator.
compl	An alternative to the ~ operator.
not	An alternative to the ! operator.
not_eq	An alternative to the != operator.
or	An alternative to the || operator.
or_eq	An alternative to the |= operator.
xor	An alternative to the ^ operator.
xor_eq	An alternative to the ^= operator.

See also

Internationalization
Universal C runtime routines by category

Single-Byte and Multibyte Character Sets

3/11/2019 • 2 minutes to read • Edit Online

The ASCII character set defines characters in the range 0x00 - 0x7F. There are a number of other character sets, primarily European, that define the characters within the range 0x00 - 0x7F identically to the ASCII character set and also define an extended character set from 0x80 - 0xFF. Thus an 8-bit, single-byte-character set (SBCS) is sufficient to represent the ASCII character set as well as the character sets for many European languages. However, some non-European character sets, such as Japanese Kanji, include many more characters than can be represented in a single-byte coding scheme, and therefore require multibyte-character set (MBCS) encoding.

> **NOTE**
>
> Many SBCS routines in the Microsoft run-time library handle multibyte bytes, characters, and strings as appropriate. Many multibyte-character sets define the ASCII character set as a subset. In many multibyte character sets, each character in the range 0x00 - 0x7F is identical to the character that has the same value in the ASCII character set. For example, in both ASCII and MBCS character strings, the one-byte null character ('\0') has value 0x00 and indicates the terminating null character.

A multibyte character set may consist of both one-byte and two-byte characters. Thus a multibyte-character string may contain a mixture of single-byte and double-byte characters. A two-byte multibyte character has a lead byte and a trail byte. In a particular multibyte-character set, the lead bytes fall within a certain range, as do the trail bytes. When these ranges overlap, it may be necessary to evaluate the particular context to determine whether a given byte is functioning as a lead byte or a trail byte.

See also

Internationalization
Universal C runtime routines by category

SBCS and MBCS Data Types

3/11/2019 • 2 minutes to read • Edit Online

Any Microsoft MBCS run-time library routine that handles only one multibyte character or one byte of a multibyte character expects an `unsigned int` argument (where 0x00 <= character value <= 0xFFFF and 0x00 <= byte value <= 0xFF). An MBCS routine that handles multibyte bytes or characters in a string context expects a multibyte-character string to be represented as an `unsigned char` pointer.

Caution

Each byte of a multibyte character can be represented in an 8-bit **char**. However, an SBCS or MBCS single-byte character of type **char** with a value greater than 0x7F is negative. When such a character is converted directly to an **int** or a **long**, the result is sign-extended by the compiler and can therefore yield unexpected results.

Therefore it is best to represent a byte of a multibyte character as an 8-bit `unsigned char`. Or, to avoid a negative result, simply convert a single-byte character of type **char** to an `unsigned char` before converting it to an **int** or a **long**.

Because some SBCS string-handling functions take (signed) **char*** parameters, a type mismatch compiler warning will result when **_MBCS** is defined. There are three ways to avoid this warning, listed in order of efficiency:

1. Use the type-safe inline functions in TCHAR.H. This is the default behavior.
2. Use the direct macros in TCHAR.H by defining **_MB_MAP_DIRECT** on the command line. If you do this, you must manually match types. This is the fastest method but is not type-safe.
3. Use the type-safe statically linked library functions in TCHAR.H. To do so, define the constant **_NO_INLINING** on the command line. This is the slowest method, but the most type-safe.

See also

Internationalization
Universal C runtime routines by category

Unicode: The Wide-Character Set

3/11/2019 • 2 minutes to read • Edit Online

A wide character is a 2-byte multilingual character code. Any character in use in modern computing worldwide, including technical symbols and special publishing characters, can be represented according to the Unicode specification as a wide character. Developed and maintained by a large consortium that includes Microsoft, the Unicode standard is now widely accepted.

A wide character is of type **wchar_t**. A wide-character string is represented as a **wchar_t[]** array and is pointed to by a `wchar_t*` pointer. You can represent any ASCII character as a wide character by prefixing the letter **L** to the character. For example, L'\0' is the terminating wide (16-bit) null character. Similarly, you can represent any ASCII string literal as a wide-character string literal simply by prefixing the letter L to the ASCII literal (L"Hello").

Generally, wide characters take up more space in memory than multibyte characters but are faster to process. In addition, only one locale can be represented at a time in multibyte encoding, whereas all character sets in the world are represented simultaneously by the Unicode representation.

See also

Internationalization
Universal C runtime routines by category

Using Generic-Text Mappings

3/11/2019 • 2 minutes to read • Edit Online

Microsoft Specific

To simplify code development for various international markets, the Microsoft run-time library provides Microsoft-specific "generic-text" mappings for many data types, routines, and other objects. These mappings are defined in TCHAR.H. You can use these name mappings to write generic code that can be compiled for any of the three kinds of character sets: ASCII (SBCS), MBCS, or Unicode, depending on a manifest constant you define using a `#define` statement. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

Preprocessor Directives for Generic-Text Mappings

#DEFINE	COMPILED VERSION	EXAMPLE
`_UNICODE`	Unicode (wide-character)	`_tcsrev` maps to `_wcsrev`
`_MBCS`	Multibyte-character	`_tcsrev` maps to `_mbsrev`
None (the default: neither `_UNICODE` nor `_MBCS` defined)	SBCS (ASCII)	`_tcsrev` maps to `strrev`

For example, the generic-text function `_tcsrev`, defined in TCHAR.H, maps to `mbsrev` if `MBCS` has been defined in your program, or to `_wcsrev` if `_UNICODE` has been defined. Otherwise `_tcsrev` maps to `strrev`.

The generic-text data type `_TCHAR`, also defined in TCHAR.H, maps to type `char` if `_MBCS` is defined, to type `wchar_t` if `_UNICODE` is defined, and to type `char` if neither constant is defined. Other data type mappings are provided in TCHAR.H for programming convenience, but `_TCHAR` is the type that is most useful.

Generic-Text Data Type Mappings

GENERIC-TEXT DATA TYPE NAME	SBCS (_UNICODE, _MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_TCHAR`	`char`	`char`	`wchar_t`
`_TINT`	`int`	`int`	`wint_t`
`_TSCHAR`	`signed char`	`signed char`	`wchar_t`
`_TUCHAR`	`unsigned char`	`unsigned char`	`wchar_t`
`_TXCHAR`	`char`	`unsigned char`	`wchar_t`
`_T` or `_TEXT`	No effect (removed by preprocessor)	No effect (removed by preprocessor)	`L` (converts following character or string to its Unicode counterpart)

For a complete list of generic-text mappings of routines, variables, and other objects, see Generic-Text Mappings.

The following code fragments illustrate the use of `_TCHAR` and `_tcsrev` for mapping to the MBCS, Unicode, and SBCS models.

```
_TCHAR *RetVal, *szString;
RetVal = _tcsrev(szString);
```

If `_MBCS` has been defined, the preprocessor maps the preceding fragment to the following code:

```
char *RetVal, *szString;
RetVal = _mbsrev(szString);
```

If `_UNICODE` has been defined, the preprocessor maps the same fragment to the following code:

```
wchar_t *RetVal, *szString;
RetVal = _wcsrev(szString);
```

If neither `_MBCS` nor `_UNICODE` has been defined, the preprocessor maps the fragment to single-byte ASCII code, as follows:

```
char *RetVal, *szString;
RetVal = strrev(szString);
```

Thus you can write, maintain, and compile a single source code file to run with routines that are specific to any of the three kinds of character sets.

END Microsoft Specific

See also

Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
Routine Mappings
A Sample Generic-Text Program

A Sample Generic-Text Program

3/11/2019 • 2 minutes to read • Edit Online

Microsoft Specific

The following program, GENTEXT.C, provides a more detailed illustration of the use of generic-text mappings defined in TCHAR.H:

```
// GENTEXT.C
// use of generic-text mappings defined in TCHAR.H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>
#include <errno.h>
#include <tchar.h>

int __cdecl _tmain(int argc, _TCHAR **argv, _TCHAR **envp)
{
   _TCHAR buff[_MAX_PATH];
   _TCHAR *str = _T("Astring");
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

#ifdef _UNICODE
   printf("Unicode version\n");
#else /* _UNICODE */
#ifdef _MBCS
   printf("MBCS version\n");
#else
   printf("SBCS version\n");
#endif
#endif /* _UNICODE */

   if (_tgetcwd(buff, _MAX_PATH) == NULL)
       printf("Can't Get Current Directory - errno=%d\n", errno);
   else
       _tprintf(_T("Current Directory is '%s'\n"), buff);
   _tprintf(_T("'%s' %hs %ls:\n"), str, amsg, wmsg);
   _tprintf(_T("'%s'\n"), _tcsrev(_tcsdup(str)));
   return 0;
}
```

If `_MBCS` has been defined, GENTEXT.C maps to the following MBCS program:

```
// crt_mbcsgtxt.c

/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * MBCS version of GENTEXT.C
 */

#include <stdio.h>
#include <stdlib.h>
#include <mbstring.h>
#include <direct.h>

int __cdecl main(int argc, char **argv, char **envp)
{
   char buff[_MAX_PATH];
   char *str = "Astring";
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

   printf("MBCS version\n");

   if (_getcwd(buff, _MAX_PATH) == NULL) {
       printf("Can't Get Current Directory - errno=%d\n", errno);
   }
   else {
       printf("Current Directory is '%s'\n", buff);
   }

   printf("'%s' %hs %ls:\n", str, amsg, wmsg);
   printf("'%s'\n", _mbsrev(_mbsdup((unsigned char*) str)));
   return 0;
}
```

If `_UNICODE` has been defined, GENTEXT.C maps to the following Unicode version of the program. For more information about using `wmain` in Unicode programs as a replacement for `main`, see Using wmain in *C Language Reference*.

```
// crt_unicgtxt.c

/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * Unicode version of GENTEXT.C
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>

int __cdecl wmain(int argc, wchar_t **argv, wchar_t **envp)
{
   wchar_t buff[_MAX_PATH];
   wchar_t *str = L"Astring";
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

   printf("Unicode version\n");

   if (_wgetcwd(buff, _MAX_PATH) == NULL) {
      printf("Can't Get Current Directory - errno=%d\n", errno);
   }
   else {
       wprintf(L"Current Directory is '%s'\n", buff);
   }

   wprintf(L"'%s' %hs %ls:\n", str, amsg, wmsg);
   wprintf(L"'%s'\n", wcsrev(wcsdup(str)));
   return 0;
}
```

If neither `_MBCS` nor `_UNICODE` has been defined, GENTEXT.C maps to single-byte ASCII code, as follows:

```
// crt_sbcsgtxt.c
/*
 * Use of generic-text mappings defined in TCHAR.H
 * Generic-Text-Mapping example program
 * Single-byte (SBCS) Ascii version of GENTEXT.C
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>

int __cdecl main(int argc, char **argv, char **envp)
{
   char buff[_MAX_PATH];
   char *str = "Astring";
   char *amsg = "Reversed";
   wchar_t *wmsg = L"Is";

   printf("SBCS version\n");

   if (_getcwd(buff, _MAX_PATH) == NULL) {
       printf("Can't Get Current Directory - errno=%d\n", errno);
   }
   else {
       printf("Current Directory is '%s'\n", buff);
   }

   printf("'%s' %hs %ls:\n", str, amsg, wmsg);
   printf("'%s'\n", strrev(strdup(str)));
   return 0;
}
```

END Microsoft Specific

See also

Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
Routine Mappings
Using Generic-Text Mappings

Using TCHAR.H Data Types with _MBCS

3/12/2019 • 2 minutes to read • Edit Online

Microsoft Specific

As the table of generic-text routine mappings indicates (see Generic-Text Mappings), when the manifest constant **_MBCS** is defined, a given generic-text routine maps to one of the following kinds of routines:

- An SBCS routine that handles multibyte bytes, characters, and strings appropriately. In this case, the string arguments are expected to be of type **char***. For example, **_tprintf** maps to **printf**; the string arguments to **printf** are of type **char***. If you use the **_TCHAR** generic-text data type for your string types, the formal and actual parameter types for **printf** match because **_TCHAR*** maps to **char***.
- An MBCS-specific routine. In this case, the string arguments are expected to be of type **unsigned char***. For example, **_tcsrev** maps to **_mbsrev**, which expects and returns a string of type **unsigned char***. Again, if you use the **_TCHAR** generic-text data type for your string types, there is a potential type conflict because **_TCHAR** maps to type **char**.

Following are three solutions for preventing this type conflict (and the C compiler warnings or C++ compiler errors that would result):

- Use the default behavior. TCHAR.H provides generic-text routine prototypes for routines in the run-time libraries, as in the following example.

  ```
  char *_tcsrev(char *);
  ```

  In the default case, the prototype for **_tcsrev** maps to **_mbsrev** through a thunk in LIBC.LIB. This changes the types of the **_mbsrev** incoming parameters and outgoing return value from **_TCHAR *** (such as **char ***) to **unsigned char ***. This method ensures type matching when you are using **_TCHAR**, but it is relatively slow because of the function call overhead.

- Use function inlining by incorporating the following preprocessor statement in your code.

  ```
  #define _USE_INLINING
  ```

  This method causes an inline function thunk, provided in TCHAR.H, to map the generic-text routine directly to the appropriate MBCS routine. The following code excerpt from TCHAR.H provides an example of how this is done.

  ```
  __inline char *_tcsrev(char *_s1)
  {return (char *)_mbsrev((unsigned char *)_s1);}
  ```

  If you can use inlining, this is the best solution, because it guarantees type matching and has no additional time cost.

- Use "direct mapping" by incorporating the following preprocessor statement in your code.

  ```
  #define _MB_MAP_DIRECT
  ```

  This approach provides a fast alternative if you do not want to use the default behavior or cannot use

inlining. It causes the generic-text routine to be mapped by a macro directly to the MBCS version of the routine, as in the following example from TCHAR.H.

```
#define _tcschr _mbschr
```

When you take this approach, you must be careful to ensure that appropriate data types are used for string arguments and string return values. You can use type casting to ensure proper type matching or you can use the **_TXCHAR** generic-text data type. **_TXCHAR** maps to type **char** in SBCS code but maps to type **unsigned char** in MBCS code. For more information about generic-text macros, see Generic-Text Mappings.

END Microsoft Specific

See also

Internationalization
Universal C runtime routines by category

Memory Allocation

3/11/2019 • 2 minutes to read • Edit Online

Use these routines to allocate, free, and reallocate memory.

Memory-Allocation Routines

ROUTINE	USE
_alloca, _malloca	Allocate memory from stack
calloc	Allocate storage for array, initializing every byte in allocated block to 0
_calloc_dbg	Debug version of **calloc**; only available in the debug versions of the run-time libraries
operator delete	Free allocated block
operator delete[]	Free allocated block
_expand	Expand or shrink block of memory without moving it
_expand_dbg	Debug version of **_expand**; only available in the debug versions of the run-time libraries
free	Free allocated block
_free_dbg	Debug version of **free**; only available in the debug versions of the run-time libraries
_freea	Free allocated block from stack
_get_heap_handle	Get Win32 HANDLE of the CRT heap.
_heapadd	Add memory to heap
_heapchk	Check heap for consistency
_heapmin	Release unused memory in heap
_heapset	Fill free heap entries with specified value
_heapwalk	Return information about each entry in heap
malloc	Allocate block of memory from heap
_malloc_dbg	Debug version of **malloc**; only available in the debug versions of the run-time libraries

ROUTINE	USE
_msize	Return size of allocated block
_msize_dbg	Debug version of **_msize**; only available in the debug versions of the run-time libraries
new	Allocate block of memory from heap
new[]	Allocate block of memory from heap
_query_new_handler	Return address of current new handler routine as set by **_set_new_handler**
_query_new_mode	Return integer indicating new handler mode set by **_set_new_mode** for **malloc**
realloc	Reallocate block to new size
_realloc_dbg	Debug version of **realloc**; only available in the debug versions of the run-time libraries
_set_new_handler	Enable error-handling mechanism when **new** operator fails (to allocate memory) and enable compilation of C++ Standard Libraries
_set_new_mode	Set new handler mode for **malloc**

See also

Universal C runtime routines by category

Process and Environment Control

3/11/2019 • 4 minutes to read • Edit Online

Use the process-control routines to start, stop, and manage processes from within a program. Use the environment-control routines to get and change information about the operating-system environment.

Process and Environment Control Functions

ROUTINE	USE
abort	Abort process without flushing buffers or calling functions registered by **atexit** and **_onexit**
assert	Test for logic error
_ASSERT, _ASSERTE macros	Similar to **assert**, but only available in the debug versions of the run-time libraries
atexit	Schedule routines for execution at program termination
_beginthread, _beginthreadex	Create a new thread on a Windows operating system process
_cexit	Perform **exit** termination procedures (such as flushing buffers), then return control to calling program without terminating process
_c_exit	Perform **_exit** termination procedures, then return control to calling program without terminating process
_cwait	Wait until another process terminates
_endthread, _endthreadex	Terminate a Windows operating system thread
_execl, _wexecl	Execute new process with argument list
_execle, _wexecle	Execute new process with argument list and given environment
_execlp, _wexeclp	Execute new process using **PATH** variable and argument list
_execlpe, _wexeclpe	Execute new process using **PATH** variable, given environment, and argument list
_execv, _wexecv	Execute new process with argument array
_execve, _wexecve	Execute new process with argument array and given environment

ROUTINE	USE
_execvp, _wexecvp	Execute new process using **PATH** variable and argument array
_execvpe, _wexecvpe	Execute new process using **PATH** variable, given environment, and argument array
exit	Call functions registered by **atexit** and **_onexit**, flush all buffers, close all open files, and terminate process
_exit	Terminate process immediately without calling **atexit** or **_onexit** or flushing buffers
getenv, _wgetenv, getenv_s, _wgetenv_s	Get value of environment variable
_getpid	Get process ID number
longjmp	Restore saved stack environment; use it to execute a nonlocal **goto**
_onexit	Schedule routines for execution at program termination; use for compatibility with Microsoft C/C++ version 7.0 and earlier
_pclose	Wait for new command processor and close stream on associated pipe
perror, _wperror	Print error message
_pipe	Create pipe for reading and writing
_popen, _wpopen	Create pipe and execute command
_putenv, _wputenv, _putenv_s, _wputenv_s	Add or change value of environment variable
raise	Send signal to calling process
setjmp	Save stack environment; use to execute non local **goto**
signal	Handle interrupt signal
_spawnl, _wspawnl	Create and execute new process with specified argument list
_spawnle, _wspawnle	Create and execute new process with specified argument list and environment
_spawnlp, _wspawnlp	Create and execute new process using **PATH** variable and specified argument list
_spawnlpe, _wspawnlpe	Create and execute new process using **PATH** variable, specified environment, and argument list

ROUTINE	USE
_spawnv, _wspawnv	Create and execute new process with specified argument array
_spawnve, _wspawnve	Create and execute new process with specified environment and argument array
_spawnvp, _wspawnvp	Create and execute new process using **PATH** variable and specified argument array
_spawnvpe, _wspawnvpe	Create and execute new process using **PATH** variable, specified environment, and argument array
system, _wsystem	Execute operating-system command

In the Windows operating system, the spawned process is equivalent to the spawning process. Any process can use **_cwait** to wait for any other process for which the process ID is known.

The difference between the **_exec** and **_spawn** families is that a **_spawn** function can return control from the new process to the calling process. In a **_spawn** function, both the calling process and the new process are present in memory unless **_P_OVERLAY** is specified. In an **_exec** function, the new process overlays the calling process, so control cannot return to the calling process unless an error occurs in the attempt to start execution of the new process.

The differences among the functions in the **_exec** family, as well as among those in the **_spawn** family, involve the method of locating the file to be executed as the new process, the form in which arguments are passed to the new process, and the method of setting the environment, as shown in the following table. Use a function that passes an argument list when the number of arguments is constant or is known at compile time. Use a function that passes a pointer to an array containing the arguments when the number of arguments is to be determined at run time. The information in the following table also applies to the wide-character counterparts of the **_spawn** and **_exec** functions.

_spawn and _exec Function Families

FUNCTIONS	USE PATH VARIABLE TO LOCATE FILE	ARGUMENT-PASSING CONVENTION	ENVIRONMENT SETTINGS
_execl, **_spawnl**	No	List	Inherited from calling process
_execle, **_spawnle**	No	List	Pointer to environment table for new process passed as last argument
_execlp, **_spawnlp**	Yes	List	Inherited from calling process
_execvpe, **_spawnvpe**	Yes	Array	Pointer to environment table for new process passed as last argument
_execlpe, **_spawnlpe**	Yes	List	Pointer to environment table for new process passed as last argument

FUNCTIONS	USE PATH VARIABLE TO LOCATE FILE	ARGUMENT-PASSING CONVENTION	ENVIRONMENT SETTINGS
_execv, **_spawnv**	No	Array	Inherited from calling process
_execve, **_spawnve**	No	Array	Pointer to environment table for new process passed as last argument
_execvp, **_spawnvp**	Yes	Array	Inherited from calling process

See also

Universal C runtime routines by category

Robustness

3/11/2019 • 2 minutes to read • Edit Online

Use the following C run-time library functions to improve the robustness of your program.

Run-Time Robustness Functions

FUNCTION	USE
_set_new_handler	Transfers control to your error-handling mechanism if the **new** operator fails to allocate memory.
_set_se_translator	Handles Win32 exceptions (C structured exceptions) as C++ typed exceptions.
set_terminate	Installs your own termination function to be called by terminate.
set_unexpected	Installs your own termination function to be called by unexpected.

See also

Universal C runtime routines by category
SetUnhandledExceptionFilter

Run-Time Error Checking

3/11/2019 • 2 minutes to read • Edit Online

The C run-time library contains the functions that support run-time error checks (RTC). Run-time error checking allows you to build your program such that certain kinds of run-time errors are reported. You specify how the errors are reported and which kinds of errors are reported. For more information, see How to: Use Native Run-Time Checks.

Use the following functions to customize the way your program does run-time error checking.

Run-Time Error Checking Functions

FUNCTION	USE
_RTC_GetErrDesc	Returns a brief description of a run-time error check type.
_RTC_NumErrors	Returns the total number of errors that can be detected by run-time error checks.
_RTC_SetErrorFunc	Designates a function as the handler for reporting run-time error checks.
_RTC_SetErrorType	Associates an error that is detected by run-time error checks with a type.

See also

Universal C runtime routines by category
/RTC (Run-Time Error Checks)
runtime_checks
Debug Routines

Searching and Sorting

3/11/2019 • 2 minutes to read • Edit Online

Use the following functions for searching and sorting.

Searching and Sorting Functions

FUNCTION	SEARCH OR SORT
bsearch	Binary search
bsearch_s	A more secure version of **bsearch**
_lfind	Linear search for given value
_lfind_s	A more secure version of **_lfind**
_lsearch	Linear search for given value, which is added to array if not found
_lsearch_s	A more secure version of **_lsearch**
qsort	Quick sort
qsort_s	A more secure version of **qsort**

See also

Universal C runtime routines by category

String Manipulation (CRT)

3/11/2019 • 3 minutes to read • Edit Online

These routines operate on null-terminated single-byte character, wide-character, and multibyte-character strings. Use the buffer-manipulation routines, described in Buffer Manipulation, to work with character arrays that do not end with a null character.

String-Manipulation Routines

ROUTINE	USE
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l, _stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l, _strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l, _strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare two character strings using code page information (**_mbsicoll** and **_mbsnicoll** are case-insensitive)
_strdec, _wcsdec, _mbsdec, _mbsdec_l	Move string pointer back one character
_strinc, _wcsinc, _mbsinc, _mbsinc_l	Advance string pointer by one character
_mbsnbcat, _mbsnbcat_l, _mbsnbcat_s, _mbsnbcat_s_l	Append, at most, first *n* bytes of one character string to another
_mbsnbcmp, _mbsnbcmp_l	Compare first *n* bytes of two character strings
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l	Return number of character bytes within supplied character count
_mbsnbcpy, _mbsnbcpy_l, _mbsnbcpy_s, _mbsnbcpy_s_l	Copy *n* bytes of string
_mbsnbicmp, _mbsnbicmp_l	Compare *n* bytes of two character strings, ignoring case
_mbsnbset, _mbsnbset_l	Set first *n* bytes of character string to specified character
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l	Return number of characters within supplied byte count
_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l	Find next character in string
_strninc, _wcsninc, _mbsninc, _mbsninc_l	Advance string pointer by *n* characters
_strspnp, _wcsspnp, _mbsspnp, _mbsspnp_l	Return pointer to first character in given string that is not in another given string
_scprintf, _scprintf_l, _scwprintf, _scwprintf_l	Return the number of characters in a formatted string
_snscanf, _snscanf_l, _snwscanf, _snwscanf_l, _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l	Read formatted data of a specified length from the standard input stream.

ROUTINE	USE
sscanf, _sscanf_l, swscanf, _swscanf_l, sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l	Read formatted data of a specified length from the standard input stream.
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l, sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l, _sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l	Write formatted data to a string
strcat, wcscat, _mbscat, strcat_s, wcscat_s, _mbscat_s	Append one string to another
strchr, wcschr, _mbschr, _mbschr_l	Find first occurrence of specified character in string
strcmp, wcscmp, _mbscmp	Compare two strings
strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l, _stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l, _strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l, _strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l	Compare two strings using current locale code page information (**_stricoll**, **_wcsicoll**, **_strnicoll**, and **_wcsnicoll** are case-insensitive)
strcpy, wcscpy, _mbscpy, strcpy_s, wcscpy_s, _mbscpy_s	Copy one string to another
strcspn, wcscspn, _mbscspn, _mbscspn_l	Find first occurrence of character from specified character set in string
_strdup, _wcsdup, _mbsdup, _strdup_dbg, _wcsdup_dbg	Duplicate string
strerror, _strerror, _wcserror, __wcserror, strerror_s, _strerror_s, _wcserror_s, __wcserror_s	Map error number to message string
strftime, wcsftime, _strftime_l, _wcsftime_l	Format date-and-time string
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l	Compare two strings without regard to case
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l, strnlen, strnlen_s, wcsnlen, wcsnlen_s, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l	Find length of string
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l, _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l	Convert string to lowercase
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l, strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l	Append characters of string
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l	Compare characters of two strings
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l, strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l	Copy characters of one string to another
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l	Compare characters of two strings without regard to case

ROUTINE	USE
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l	Set first *n* characters of string to specified character
strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l	Find first occurrence of character from one string in another string
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l	Find last occurrence of given character in string
_strrev, _wcsrev, _mbsrev, _mbsrev_l	Reverse string
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l	Set all characters of string to specified character
strspn, wcsspn, _mbsspn, _mbsspn_l	Find first occurrence in a string of a character not found in another string
strstr, wcsstr, _mbsstr, _mbsstr_l	Find first occurrence of specified string in another string
strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l, strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l	Find next token in string
_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr, _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l	Convert string to uppercase
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l	Transform string into collated form based on locale-specific information
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l, vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l, _vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l	Write formatted output using a pointer to a list of arguments
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l, vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l	Write formatted output using a pointer to a list of arguments

See also

Universal C runtime routines by category

System Calls

3/11/2019 • 2 minutes to read • Edit Online

The following functions are Windows operating system calls.

System Call Functions

FUNCTION	USE
_findclose	Release resources from previous find operations
_findfirst, _findfirst32, _findfirst64, _findfirsti64, _findfirst32i64, _findfirst64i32, _wfindfirst, _wfindfirst32, _wfindfirst64, _wfindfirsti64, _wfindfirst32i64, _wfindfirst64i32	Find file with specified attributes
_findnext, _findnext32, _findnext64, _findnexti64, _findnext32i64, _findnext64i32, _wfindnext, _wfindnext32, _wfindnexti64, _wfindnext64, _wfindnexti64	Find next file with specified attributes

See also

Universal C runtime routines by category
File Handling
Directory Control
Low-Level I/O

Time Management

5/8/2019 • 2 minutes to read • Edit Online

Use these functions to get the current time and convert, adjust, and store it as necessary. The current time is the system time.

The **_ftime** and **localtime** routines use the **TZ** environment variable. If **TZ** is not set, the run-time library attempts to use the time-zone information specified by the operating system. If this information is unavailable, these functions use the default value of PST8PDT. For more information on **TZ**, see _tzset; also see _daylight, timezone, and _tzname.

Time Routines

FUNCTION	USE
asctime, _wasctime, asctime_s, _wasctime_s	Convert time from type **struct tm** to character string. The versions of these functions with the **_s** suffix are more secure.
clock	Return elapsed wall-clock time for process.
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, _ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s	Convert time from type **time_t**, **__time32_t** or **__time64_t** to character string. The versions of these functions with the **_s** suffix are more secure.
difftime, _difftime32, _difftime64	Compute difference between two times.
_ftime, _ftime32, _ftime64,_ftime_s, _ftime32_s, _ftime64_s	Store current system time in variable of type **struct _timeb** or type **struct __timeb64** The versions of these functions with the **_s** suffix are more secure.
_futime, _futime32, _futime64	Set modification time on open file
gmtime, _gmtime32, _gmtime64, gmtime_s, _gmtime32_s, _gmtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm**.The versions of these functions with the **_s** suffix are more secure.
localtime, _localtime32, _localtime64, localtime_s, _localtime32_s, _localtime64_s	Convert time from type **time_t** to **struct tm** or from type **__time64_t** to **struct tm** with local correction. The versions of these functions with the **_s** suffix are more secure.
_mkgmtime, _mkgmtime32, _mkgmtime64	Convert time to calendar value in Greenwich Mean Time.
mktime, _mktime32, _mktime64	Convert time to calendar value.
_strdate, _wstrdate, _strdate_s, _wstrdate_s	Return current system date as string. The versions of these functions with the **_s** suffix are more secure.
strftime, wcsftime, _strftime_l, _wcsftime_l	Format date-and-time string for international use.
_strtime, _wstrtime, _strtime_s, _wstrtime_s	Return current system time as string. The versions of these functions with the **_s** suffix are more secure.

FUNCTION	USE
time, _time32, _time64	Get current system time as type **time_t**, **__time32_t** or as type **__time64_t**.
_tzset	Set external time variables from environment time variable **TZ**.
_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64	Set modification time for specified file using either current time or time value stored in structure.

NOTE

In all versions of Microsoft C/C++ except Microsoft C/C++ version 7.0, and in all versions of Visual C++, the time function returns the current time as the number of seconds elapsed since midnight on January 1, 1970. In Microsoft C/C++ version 7.0, **time** returned the current time as the number of seconds elapsed since midnight on December 31, 1899.

NOTE

In versions of Visual C++ and Microsoft C/C++ before Visual Studio 2005, **time_t** was a **long int** (32 bits) and hence could not be used for dates past 3:14:07 January 19, 2038, UTC. **time_t** is now equivalent to **__time64_t** by default, but defining **_USE_32BIT_TIME_T** changes **time_t** to **__time32_t** and forces many time functions to call versions that take the 32-bit **time_t**. For more information, see Standard Types and comments in the documentation for the individual time functions.

See also

Universal C runtime routines by category

Windows Runtime Unsupported CRT Functions

3/11/2019 • 2 minutes to read • Edit Online

Many C run-time (CRT) APIs can't be used in Universal Windows Platform (UWP) apps that execute in the Windows Runtime. These apps are built by using the /ZW compiler flag. For a list of unsupported CRT functions, see CRT functions not supported in Universal Windows Platform apps.

All CRT APIs are described in the Alphabetical Function Reference section of the documentation.

See also

Universal C runtime routines by category
Alphabetical Function Reference

Internal CRT Globals and Functions

3/11/2019 • 2 minutes to read • Edit Online

The C runtime (CRT) library contains functions and global variables that are used only to support the public library interface. Some of them are exposed in public headers as implementation details. Although these functions and global variables are accessible through public exports, they are not intended for use by your code. We recommend that you change any code that uses these functions and variables to use public library equivalents instead. These functions may change from version to version. They are listed here to help you identify them. Links are provided when additional documentation exists, but in general, these implementation details are not documented.

Internal CRT Globals and Value Macros

These global variables and macro definitions are used to implement the CRT.

NAME
__badioinfo
_acmdln
_commode
_crtAssertBusy
_crtBreakAlloc
__initenv
__lconv
__mb_cur_max
__pioinfo
__unguarded_readlc_active
_wcmdln
__winitenv

Internal CRT Functions and Function Macros

These functions and function macros are used to implement the CRT and the C++ Standard Library.

NAME
__acrt_iob_func
__AdjustPointer

NAME
_assert
__BuildCatchObject
__BuildCatchObjectHelper
__C_specific_handler
_calloc_base
_chkesp
__chkstk
_chkstk
_chvalidator
_chvalidator_l
_CIacos
_CIasin
_CIatan
_CIatan2
_CIcos
_CIcosh
_CIexp
_CIfmod
_CIlog
_CIlog10
_CIpow
_CIsin
_CIsinh
_CIsqrt
_CItan

NAME
_CItanh
__clean_type_info_names_internal
_configure_narrow_argv
_configure_wide_argv
__conio_common_vcprintf
__conio_common_vcprintf_p
__conio_common_vcprintf_s
__conio_common_vcscanf
__conio_common_vcwprintf
__conio_common_vcwprintf_p
__conio_common_vcwprintf_s
__conio_common_vcwscanf
__CppXcptFilter
__create_locale
_crt_atexit
_crt_at_quick_exit
__crtCompareStringA
__crtCompareStringEx
__crtCompareStringW
__crtCreateEventExW
__crtCreateSemaphoreExW
__crtCreateSymbolicLinkW
_crt_debugger_hook
__crtEnumSystemLocalesEx
__crtFlsAlloc

NAME
__crtFlsFree
__crtFlsGetValue
__crtFlsSetValue
_CrtGetCheckCount
__crtGetDateFormatEx
__crtGetFileInformationByHandleEx
__crtGetLocaleInfoEx
__crtGetShowWindowMode
__crtGetTickCount64
__crtGetTimeFormatEx
__crtGetUserDefaultLocaleName
__crtInitializeCriticalSectionEx
__crtIsPackagedApp
__crtIsValidLocaleName
__crtLCMapStringA
__crtLCMapStringEx
__crtLCMapStringW
_CrtSetCheckCount
_CrtSetDbgBlockType
__crtSetFileInformationByHandle
__crtSetThreadStackGuarantee
__crtSetUnhandledExceptionFilter
__crtSleep
__crtTerminateProcess
__crtUnhandledException

NAME
__CxxDetectRethrow
__CxxExceptionFilter
__CxxFrameHandler
__CxxFrameHandler2
__CxxFrameHandler3
__CxxLongjmpUnwind
__CxxQueryExceptionSize
__CxxRegisterExceptionObject
_CxxThrowException
__CxxUnregisterExceptionObject
__daylight
_dclass
__DestructExceptionObject
__dllonexit
__doserrno
_dosmaperr
_dpcomp
_dsign
__dstbias
_dtest
_EH_prolog
_errno
_except_handler2
_except_handler3
_except_handler4_common

NAME
_except1
_execute_onexit_table
_fdclass
_fdpcomp
_fdsign
_fdtest
_filbuf
_FindAndUnlinkFrame
_flsbuf
__fpe_flt_rounds
_FPE_Raise
__fpecode
__FrameUnwindFilter
_fread_nolock_s
_free_base
__free_locale
_freea_s
_freefls
_ftol
__get_current_locale
__get_flsindex
_get_initial_narrow_environment
_get_initial_wide_environment
_get_narrow_winmain_command_line
_get_stream_buffer_pointers

NAME
__get_tlsindex
_get_wide_winmain_command_line
_Getdays
__getmainargs
_Getmonths
__GetPlatformExceptionInfo
_getptd
_Gettnames
_global_unwind2
_inconsistency
_initialize_lconv_for_unsigned_char
_initialize_narrow_environment
_initialize_onexit_table
_initialize_wide_environment
_initptd
_invalid_parameter
_invoke_watson
__iob_func
_IsExceptionObjectToBeDestroyed
___lc_codepage_func
___lc_collate_cp_func
___lc_locale_name_func
__lconv_init
_ldclass
_ldpcomp

NAME
_ldsign
_ldtest
__libm_sse2_acos
_libm_sse2_acos_precise
__libm_sse2_acosf
__libm_sse2_asin
_libm_sse2_asin_precise
__libm_sse2_asinf
__libm_sse2_atan
_libm_sse2_atan_precise
__libm_sse2_atan2
__libm_sse2_atanf
__libm_sse2_cos
_libm_sse2_cos_precise
__libm_sse2_cosf
__libm_sse2_exp
_libm_sse2_exp_precise
__libm_sse2_expf
__libm_sse2_log
_libm_sse2_log_precise
__libm_sse2_log10
_libm_sse2_log10_precise
__libm_sse2_log10f
__libm_sse2_logf
__libm_sse2_pow

NAME
_libm_sse2_pow_precise
__libm_sse2_powf
__libm_sse2_sin
_libm_sse2_sin_precise
__libm_sse2_sinf
_libm_sse2_sqrt_precise
__libm_sse2_tan
_libm_sse2_tan_precise
__libm_sse2_tanf
_local_unwind2
_local_unwind4
_lock_locales
_longjmpex
_malloc_base
___mb_cur_max_func
___mb_cur_max_l_func
_mbctype
_NLG_Dispatch2
_NLG_Return
_NLG_Return2
__p___argc
__p___argv
__p___initenv
__p___mb_cur_max
__p___wargv

NAME
__p___winitenv
__p__acmdln
__p__commode
__p__crtAssertBusy
__p__crtBreakAlloc
__p__crtDbgFlag
__p__daylight
__p__dstbias
__p__environ
__p__fmode
__p__iob
__p__mbcasemap
__p__mbctype
__p__pctype
__p__pgmptr
__p__pwctype
__p__timezone
__p__tzname
__p__wcmdln
__p__wenviron
__p__wpgmptr
_pctype
__pctype_func
_pwctype
__pwctype_func

NAME
__pxcptinfoptrs
_query_app_type
_realloc_base
_register_onexit_function
_register_thread_local_exe_atexit_callback
__report_gsfailure
__RTCastToVoid
__RTDynamicCast
__RTtypeid
_seh_filter_dll
_seh_filter_exe
_seh_longjmp_unwind
_seh_longjmp_unwind4
__set_app_type
_set_malloc_crt_max_wait
_setjmp3
__setlc_active
___setlc_active_func
__setusermatherr
_SetWinRTOutOfMemoryExceptionCallback
_sopen_dispatch
__std_exception_copy
__std_exception_destroy
__std_type_info_destroy_list
__stdio_common_vfprintf

NAME
__stdio_common_vfprintf_p
__stdio_common_vfprintf_s
__stdio_common_vfscanf
__stdio_common_vfwprintf
__stdio_common_vfwprintf_p
__stdio_common_vfwprintf_s
__stdio_common_vfwscanf
__stdio_common_vsnprintf_s
__stdio_common_vsnwprintf_s
__stdio_common_vsprintf
__stdio_common_vsprintf_p
__stdio_common_vsprintf_s
__stdio_common_vsscanf
__stdio_common_vswprintf
__stdio_common_vswprintf_p
__stdio_common_vswprintf_s
__stdio_common_vswscanf
_Strftime
__STRINGTOLD
__STRINGTOLD_L
__strncnt
__sys_errlist
__sys_nerr
__threadhandle
__threadid

NAME
__timezone
__TypeMatch
__tzname
__unDName
__unDNameEx
__unDNameHelper
__unguarded_readlc_active
___unguarded_readlc_active_add_func
_unloaddll
_unlock_locales
_vacopy
_ValidateExecute
_ValidateRead
_ValidateWrite
_VCrtDbgReportA
_VCrtDbgReportW
_W_Getdays
_W_Getmonths
_W_Getnames
_wassert
_Wcsftime
__wcsncnt
__wgetmainargs
_wsopen_dispatch
_Xbad_alloc

NAME
_Xlength_error

See also

Universal C runtime routines by category

_abnormal_termination

3/11/2019 • 2 minutes to read • Edit Online

Indicates whether the `__finally` block of a try-finally statement is entered while the system is executing an internal list of termination handlers.

Syntax

```
int   _abnormal_termination(
   );
```

Return Value

true if the system is *unwinding* the stack; otherwise, **false**.

Remarks

This is an internal function used to manage unwinding exceptions, and is not intended to be called from user code.

Requirements

ROUTINE	REQUIRED HEADER
_abnormal_termination	excpt.h

See also

try-finally Statement

_acmdln, _tcmdln, _wcmdln

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT global variable. The command line.

Syntax

```
char * _acmdln;
wchar_t * _wcmdln;

#ifdef WPRFLAG
   #define _tcmdln _wcmdln
#else
   #define _tcmdln _acmdln
```

Remarks

These CRT internal variables store the complete command line. They are exposed in the exported symbols for the CRT, but are not intended for use in your code. `_acmdln` stores the data as a character string. `_wcmdln` stores the data as a wide character string. `_tcmdln` can be defined as either `_acmdln` or `_wcmdln`, depending on which is appropriate.

See also

Global Variables

_CIatan

3/11/2019 • 2 minutes to read • Edit Online

Calculates the arctangent of the top value on the stack.

Syntax

```
void __cdecl _CIatan();
```

Remarks

This version of the `atan` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
atan, atanf, atanl, atan2, atan2f, atan2l

_CIatan2

3/11/2019 • 2 minutes to read • Edit Online

Calculates the arctangent of *x* / *y* where *x* and *y* are values on the top of the stack.

Syntax

```
void __cdecl _CIatan2();
```

Remarks

This version of the `atan2` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
atan, atanf, atanl, atan2, atan2f, atan2l

_CIcos

2/4/2019 • 2 minutes to read • Edit Online

Calculates the cosine of the top value in the floating-point stack.

Syntax

```
void __cdecl _CIcos();
```

Remarks

This version of the cos function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the floating-point stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
cos, cosf, cosl

_CIexp

3/11/2019 • 2 minutes to read • Edit Online

Calculates the exponential of the top value on the stack.

Syntax

```
void __cdecl _CIexp();
```

Remarks

This version of the `exp` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
exp, expf, expl

_CIfmod

3/11/2019 • 2 minutes to read • Edit Online

Calculates the floating-point remainder of the top two values on the stack.

Syntax

```
void __cdecl _CIfmod();
```

Remarks

This version of the `fmod` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
fmod, fmodf

_CIlog

3/11/2019 • 2 minutes to read • Edit Online

Calculates the natural logarithm of the top value in the stack.

Syntax

```
void __cdecl _CIlog();
```

Remarks

This version of the `log` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
log, logf, log10, log10f

_CIlog10

3/11/2019 • 2 minutes to read • Edit Online

Performs a `log10` operation on the top value in the stack.

Syntax

```
void __cdecl _CIlog10();
```

Remarks

This version of the `log10` function has a specialized calling convention that the compiler understands. The function speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
log, logf, log10, log10f

_CIpow

3/11/2019 • 2 minutes to read • Edit Online

Calculates *x* raised to the *y* power based on the top values in the stack.

Syntax

```
void __cdecl _CIpow();
```

Remarks

This version of the `pow` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
pow, powf, powl

_CIsin

2/4/2019 • 2 minutes to read • Edit Online

Calculates the sine of the top value in the floating-point stack.

Syntax

```
void __cdecl _CIsin();
```

Remarks

This intrinsic version of the sin function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the floating-point stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
sin, sinf, sinl

_CIsqrt

3/11/2019 • 2 minutes to read • Edit Online

Calculates the square root of the top value in the stack.

Syntax

```
void __cdecl _CIsqrt();
```

Remarks

This version of the `sqrt` function has a specialized calling convention that the compiler understands. It speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
sqrt, sqrtf, sqrtl

_CItan

2/4/2019 • 2 minutes to read • Edit Online

Calculates the tangent of the top value on the floating-point stack.

Syntax

```
void __cdecl _CItan();
```

Remarks

This version of the tan function has a specialized calling convention that the compiler understands. The function speeds up the execution because it prevents copies from being generated and helps with register allocation.

The resulting value is pushed onto the top of the floating-point stack.

Requirements

Platform: x86

See also

Alphabetical Function Reference
tan, tanf, tanl

__crtLCMapStringW

10/31/2018 • 2 minutes to read • Edit Online

Maps one character string to another, performing a specified locale-dependent transformation. This function can also be used to generate a sort key for the input string.

Syntax

```
int __crtLCMapStringW(
   LCID    Locale,
   DWORD   dwMapFlags,
   LPCWSTR lpSrcStr,
   int     cchSrc,
   LPWSTR  lpDestStr,
   int     cchDest)
```

Parameters

Locale
Locale identifier. The locale provides a context for the string mapping or sort key generation. An application can use the `MAKELCID` macro to create a locale identifier.

dwMapFlags
The type of transformation to be used during string mapping or sort key generation.

lpSrcStr
Pointer to a source string that the function maps or uses for sort key generation. This parameter is assumed to be a Unicode string.

cchSrc
Size, in characters, of the string pointed to by the `lpSrcStr` parameter. This count can include the null terminator, or not include it.

A `cchSrc` value of -1 specifies that the string pointed to by `lpSrcStr` is null-terminated. If this is the case, and this function is being used in its string-mapping mode, the function calculates the string's length itself, and null-terminates the mapped string stored into `*lpDestStr`.

lpDestStr
Long pointer to a buffer into which the function stores the mapped string or sort key.

cchDest
Size, in characters, of the buffer pointed to by `lpDestStr`.

Return Value

If the value of `cchDest` is nonzero, the number of characters, or bytes if `LCMAP_SORTKEY` is specified, written to the buffer indicates success. This count includes room for a null terminator.

If the value of `cchDest` is zero, the size of the buffer in characters, or bytes if `LCMAP_SORTKEY` is specified, required to receive the translated string or sort key indicates success. This size includes room for a null terminator.

Zero indicates failure. To get extended error information, call the `GetLastError` function.

Remarks

If `cchSrc` is greater than zero and `lpSrcStr` is a null-terminated string, `__crtLCMapStringW` sets `cchSrc` to the length of the string. Then `__crtLCMapStringW` calls the wide string (Unicode) version of the `LCMapString` function with the specified parameters. For more information about the parameters and return value of this function, see the LCMapString.

Requirements

ROUTINE	REQUIRED HEADER
__crtLCMapStringW	awint.h

__CxxFrameHandler

10/31/2018 • 2 minutes to read • Edit Online

Internal CRT function. Used by the CRT to handle structured exception frames.

Syntax

```
EXCEPTION_DISPOSITION __CxxFrameHandler(
      EHExceptionRecord  *pExcept,
      EHRegistrationNode *pRN,
      void               *pContext,
      DispatcherContext  *pDC
   )
```

Parameters

pExcept
Exception record that is passed to the possible `catch` statements.

pRN
Dynamic information about the stack frame that is used to handle the exception. For more information, see ehdata.h.

pContext
Context. (Not used on Intel processors.)

pDC
Additional information about the function entry and stack frame.

Return Value

One of the *filter expression* values used by the try-except Statement.

Remarks

Requirements

ROUTINE	REQUIRED HEADER
__CxxFrameHandler	excpt.h, ehdata.h

__dllonexit

3/11/2019 • 2 minutes to read • Edit Online

Registers a routine to be called at exit time.

Syntax

```
_onexit_t __dllonexit(   _onexit_t func,
   _PVFV **  pbegin,
   _PVFV **  pend
   )
```

Parameters

func
Pointer to a function to be executed upon exit.

pbegin
Pointer to a variable that points to the beginning of a list of functions to execute on detach.

pend
Pointer to variable that points to the end of a list of functions to execute on detach.

Return Value

If successful, a pointer to the user's function. Otherwise, a **NULL** pointer.

Remarks

The `__dllonexit` function is analogous to the _onexit function except that the global variables used by that function are not visible to this routine. Instead of global variables, this function uses the `pbegin` and `pend` parameters.

The `_onexit` and `atexit` functions in a DLL linked with MSVCRT.LIB must maintain their own atexit/_onexit list. This routine is the worker that gets called by such DLLs.

The `_PVFV` type is defined as `typedef void (__cdecl *_PVFV)(void)`.

Requirements

ROUTINE	REQUIRED FILE
__dllonexit	onexit.c

See also

_onexit, _onexit_m

_except_handler3

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. Used by a framework to find the appropriate exception handler to process the current exception.

Syntax

```
int _except_handler3(
   PEXCEPTION_RECORD exception_record,
   PEXCEPTION_REGISTRATION registration,
   PCONTEXT context,
   PEXCEPTION_REGISTRATION dispatcher
);
```

Parameters

exception_record
[in] Information about the specific exception.

registration
[in] The record that indicates which scope table should be used to find the exception handler.

context
[in] Reserved.

dispatcher
[in] Reserved.

Return Value

If an exception should be dismissed, returns `DISPOSITION_DISMISS`. If the exception should be passed up a level to the encapsulating exception handlers, returns `DISPOSITION_CONTINUE_SEARCH`.

Remarks

If this method finds an appropriate exception handler, it passes the exception to the handler. In this situation, this method does not return to the code that called it and the return value is irrelevant.

See also

Alphabetical Function Reference

_execute_onexit_table, _initialize_onexit_table, _register_onexit_function

3/11/2019 • 2 minutes to read • Edit Online

Manages the routines to be called at exit time.

Syntax

```
int _initialize_onexit_table(
    _onexit_table_t* table
    );

int _register_onexit_function(
    _onexit_table_t* table,
    _onexit_t        function
    );

int _execute_onexit_table(
    _onexit_table_t* table
    );
```

Parameters

table
[in, out] Pointer to the onexit function table.

function
[in] Pointer to a function to add to the onexit function table.

Return Value

If successful, returns 0. Otherwise, returns a negative value.

Remarks

These functions are infrastructure implementation details used to support the C runtime, and should not be called directly from your code. The C runtime uses an *onexit function table* to represent the sequence of functions registered by calls to `atexit`, `at_quick_exit`, and `_onexit`. The onexit function table data structure is an opaque implementation detail of the C runtime; the order and meaning of its data members may change. They should not be inspected by external code.

The `_initialize_onexit_table` function initializes the onexit function table to its initial value. This function must be called before the onexit function table is passed to either `_register_onexit_function` or `_execute_onexit_table`.

The `_register_onexit_function` function appends a function to the end of the onexit function table.

The `_execute_onexit_table` function executes all of the functions in the onexit function table, clears the table, and then returns. After a call to `_execute_onexit_table`, the table is in a non-valid state; it must be reinitialized by a call to `_initialize_onexit_table` before it is used again.

Requirements

ROUTINE	REQUIRED HEADER
`_initialize_onexit_table function`, `_register_onexit_function`, `_execute_onexit_table`	C, C++: <process.h>

The `_initialize_onexit_table`, `_register_onexit_function`, and `_execute_onexit_table` functions are Microsoft specific. For compatibility information, see Compatibility.

See also

atexit
exit, _Exit, _exit
_onexit, _onexit_m

__getmainargs, __wgetmainargs

11/9/2018 • 2 minutes to read • Edit Online

Invokes command-line parsing and copies the arguments to `main()` back through the passed pointers.

Syntax

```
int __getmainargs(
    int * _Argc,
   char *** _Argv,
   char *** _Env,
   int _DoWildCard,
_startupinfo * _StartInfo);

int __wgetmainargs (
   int *_Argc,
   wchar_t ***_Argv,
   wchar_t ***_Env,
   int _DoWildCard,
   _startupinfo * _StartInfo)
```

Parameters

`_Argc`

An integer that contains the number of arguments that follow in `argv`. The `argc` parameter is always greater than or equal to 1.

`_Argv`

An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, `argv[0]` is the command with which the program is invoked, argv[1] is the first command-line argument, and so on, until argv[argc], which is always **NULL**. The first command-line argument is always `argv[1]` and the last one is `argv[argc - 1]`.

`_Env`

An array of strings that represent the variables set in the user's environment. This array is terminated by a **NULL** entry.

`_DoWildCard`

An integer that if set to 1 expands the wildcards in the command line arguments, or if set to 0 does nothing.

`_StartInfo`

Other information to be passed to the CRT DLL.

Return Value

0 if successful; a negative value if unsuccessful.

Remarks

Use `__getmainargs` on non-wide character platforms, and `__wgetmainargs` on wide-character (Unicode) platforms.

Requirements

ROUTINE	REQUIRED HEADER
__getmainargs	internal.h
__wgetmainargs	internal.h

___lc_codepage_func

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. Retrieves the current code page of the thread.

Syntax

```
UINT ___lc_codepage_func(void);
```

Return Value

The current code page of the thread.

Remarks

`___lc_codepage_func` is an internal CRT function that is used by other CRT functions to get the current code page from the thread local storage for CRT data. This information is also available by using the _get_current_locale function.

A *code page* is a mapping of single-byte or double-byte codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages. For more information about code pages, see Code Pages.

Internal CRT functions are implementation-specific and subject to change with each release. We don't recommend their use in your code.

Requirements

ROUTINE	REQUIRED HEADER
`___lc_codepage_func`	crt\src\setlocal.h

See also

_get_current_locale
setlocale, _wsetlocale
_create_locale, _wcreate_locale
_free_locale

___lc_collate_cp_func

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. Retrieves the current collation code page of the thread.

Syntax

```
UINT ___lc_codepage_func(void);
```

Return Value

The current collation code page of the thread.

Remarks

`___lc_collate_cp_func` is an internal CRT function that is used by other CRT functions to get the current collation code page from the thread local storage for CRT data. This information is also available by using the _get_current_locale function.

Internal CRT functions are implementation-specific and subject to change with each release. We don't recommend their use in your code.

Requirements

ROUTINE	REQUIRED HEADER
`___lc_collate_cp_func`	crt\src\setlocal.h

See also

_get_current_locale
setlocale, _wsetlocale
_create_locale, _wcreate_locale
_free_locale

___lc_locale_name_func

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. Retrieves the current locale name of the thread.

Syntax

```
wchar_t** ___lc_locale_name_func(void);
```

Return Value

A pointer to a string that contains the current locale name of the thread.

Remarks

`___lc_locale_name_func` is an internal CRT function that is used by other CRT functions to get the current locale name from the thread local storage for CRT data. This information is also available by using the _get_current_locale function or the setlocale, _wsetlocale functions.

Internal CRT functions are implementation-specific and subject to change with each release. We don't recommend their use in your code.

Requirements

ROUTINE	REQUIRED HEADER
`___lc_locale_name_func`	crt\src\setlocal.h

See also

_get_current_locale
setlocale, _wsetlocale
_create_locale, _wcreate_locale
_free_locale

_local_unwind2

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT Function. Runs all termination handlers that are listed in the indicated scope table.

Syntax

```
void _local_unwind2(
   PEXCEPTION_REGISTRATION xr,
   int stop
);
```

Parameters

xr
[in] A registration record that is associated with one scope table.

stop
[in] The lexical level that indicates where `_local_unwind2` should stop.

Remarks

This method is used only by the run-time environment. Do not call the method in your code.

When this method executes termination handlers, it starts at the current lexical level and works its way up in lexical levels until it reaches the level that is indicated by `stop`. It does not execute termination handlers at the level that is indicated by `stop`.

See also

Alphabetical Function Reference

___mb_cur_max_func, ___mb_cur_max_l_func, __p___mb_cur_max, __mb_cur_max

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. Retrieves the maximum number of bytes in a multibyte character for the current or specified locale.

Syntax

```
int ___mb_cur_max_func(void);
int ___mb_cur_max_l_func(_locale_t locale);
int * __p___mb_cur_max(void);
#define __mb_cur_max (___mb_cur_max_func())
```

Parameters

locale The locale structure to retrieve the result from. If this value is null, the current thread locale is used.

Return Value

The maximum number of bytes in a multibyte character for the current thread locale or the specified locale.

Remarks

This is an internal function that the CRT uses to retrieve the current value of the MB_CUR_MAX macro from thread local storage. We recommend that you use the `MB_CUR_MAX` macro in your code for portability.

The `__mb_cur_max` macro is a convenient way to call the `___mb_cur_max_func()` function. The `__p___mb_cur_max` function is defined for compatibility with Visual C++ 5.0 and earlier versions.

Internal CRT functions are implementation-specific and subject to change with each release. We don't recommend their use in your code.

Requirements

ROUTINE	REQUIRED HEADER
`___mb_cur_max_func`, `___mb_cur_max_l_func`, `__p___mb_cur_max`	<ctype.h>, <stdlib.h>

See also

MB_CUR_MAX

__p__commode

10/31/2018 • 2 minutes to read • Edit Online

Points to the `_commode` global variable, which specifies the default *file commit mode* for file I/O operations.

Syntax

```
int * __p__commode(
   );
```

Return Value

Pointer to the `_commode` global variable.

Remarks

The `__p__commode` function is for internal use only, and should not be called from user code.

File commit mode specifies when critical data is written to disk. For more information, see fflush.

Requirements

ROUTINE	REQUIRED HEADER
__p__commode	internal.h

__p__fmode

2/4/2019 • 2 minutes to read • Edit Online

Points to the `_fmode` global variable, which specifies the default *file translation mode* for file I/O operations.

Syntax

```
int* __p__fmode(
   );
```

Return Value

Pointer to the `_fmode` global variable.

Remarks

The `__p__fmode` function is for internal use only, and should not be called from user code.

File translation mode specifies either `binary` or `text` translation for _open and _pipe I/O operations. For more information, see _fmode.

Requirements

ROUTINE	REQUIRED HEADER
__p__fmode	stdlib.h

__pctype_func

3/11/2019 • 2 minutes to read • Edit Online

Retrieves a pointer to an array of character classification information.

Syntax

```
const unsigned short *__pctype_func(
   )
```

Return Value

A pointer to an array of character classification information.

Remarks

The information in the character classification table is for internal use only, and is used by various functions that classify characters of type `char`. For more information, see the `Remarks` section of _pctype, _pwctype, _wctype, _mbctype, _mbcasemap.

Requirements

ROUTINE	REQUIRED HEADER
__pctype_func	ctype.h

See also

_pctype, _pwctype, _wctype, _mbctype, _mbcasemap

__RTDynamicCast

10/31/2018 • 2 minutes to read • Edit Online

Runtime implementation of the dynamic_cast operator.

Syntax

```
PVOID __RTDynamicCast (
   PVOID inptr,
   LONG VfDelta,
   PVOID SrcType,
   PVOID TargetType,
   BOOL isReference
   ) throw(...)
```

Parameters

inptr
Pointer to a polymorphic object.

VfDelta
Offset of virtual function pointer in object.

SrcType
Static type of object pointed to by the `inptr` parameter.

TargetType
Intended result of cast.

isReference
true if input is a reference; **false** if input is a pointer.

Return Value

Pointer to the appropriate sub-object, if successful; otherwise, **NULL**.

Exceptions

`bad_cast()` if the input to `dynamic_cast<>` is a reference and the cast fails.

Remarks

Converts `inptr` to an object of type `TargetType`. The type of `inptr` must be a pointer if `TargetType` is a pointer, or an l-value if `TargetType` is a reference. `TargetType` must be a pointer or a reference to a previously defined class type, or a pointer to void.

Requirements

ROUTINE	REQUIRED HEADER
__RTDynamicCast	rtti.h

__set_app_type

10/31/2018 • 2 minutes to read • Edit Online

Sets the current application type.

Syntax

```
void __set_app_type (
   int at
   )
```

Parameters

at

A value that indicates the application type. The possible values are:

VALUE	DESCRIPTION
_UNKNOWN_APP	Unknown application type.
_CONSOLE_APP	Console (command-line) application.
_GUI_APP	GUI (Windows) application.

Remarks

Requirements

ROUTINE	REQUIRED HEADER
__set_app_type	internal.h

_set_app_type

3/11/2019 • 2 minutes to read • Edit Online

An internal function used at startup to tell the CRT whether the app is a console app or a GUI app.

Syntax

```
typedef enum _crt_app_type
{
    _crt_unknown_app,
    _crt_console_app,
    _crt_gui_app
} _crt_app_type;

void __cdecl _set_app_type(
    _crt_app_type appType
    );
```

Parameters

appType
A value that indicates the application type. The possible values are:

VALUE	DESCRIPTION
_crt_unknown_app	Unknown application type.
_crt_console_app	Console (command-line) application.
_crt_gui_app	GUI (Windows) application.

Remarks

Normally, you do not need to call this function. It is part of the C runtime startup code that executes before `main` is called in your app.

Requirements

ROUTINE	REQUIRED HEADER
_set_app_type	process.h

_setjmp3

3/11/2019 • 2 minutes to read • Edit Online

Internal CRT function. A new implementation of the `setjmp` function.

Syntax

```
int _setjmp3(
   OUT jmp_buf env,
   int count,
   (optional parameters)
);
```

Parameters

env
[out] Address of the buffer for storing state information.

count
[in] The number of additional `DWORD`s of information that are stored in the `optional parameters`.

optional parameters
[in] Additional data pushed down by the `setjmp` intrinsic. The first `DWORD` is a function pointer that is used to unwind extra data and return to a nonvolatile register state. The second `DWORD` is the try level to be restored. Any further data is saved in the generic data array in the `jmp_buf`.

Return Value

Always returns 0.

Remarks

Do not use this function in a C++ program. It is an intrinsic function that does not support C++. For more information about how to use `setjmp`, see Using setjmp/longjmp.

Requirements

See also

Alphabetical Function Reference
setjmp

___setlc_active_func, ___unguarded_readlc_active_add_func

3/11/2019 • 2 minutes to read • Edit Online

OBSOLETE. The CRT exports these internal functions only to preserve binary compatibility.

Syntax

```
int ___setlc_active_func(void);
int * ___unguarded_readlc_active_add_func(void);
```

Return Value

The value returned is not significant.

Remarks

Although the internal CRT functions `___setlc_active_func` and `___unguarded_readlc_active_add_func` are obsolete and no longer used, they are exported by the CRT library to preserve binary compatibility. The original purpose of `___setlc_active_func` was to return the number of currently active calls to the `setlocale` function. The original purpose of `___unguarded_readlc_active_add_func` was to return the number of functions that referenced the locale without locking it.

Requirements

ROUTINE	REQUIRED HEADER
`___setlc_active_func`, `___unguarded_readlc_active_add_func`	none

See also

setlocale, _wsetlocale

__setusermatherr

10/31/2018 • 2 minutes to read • Edit Online

Specifies a user-supplied rountine to handle math errors, instead of the _matherr routine.

Syntax

```
void __setusermatherr(
   _HANDLE_MATH_ERROR pf
   )
```

Parameters

pf
Pointer to an implementation of `_matherr` that is supplied by the user.

The type of the *pf* parameter is declared as `typedef int (__cdecl * _HANDLE_MATH_ERROR)(struct _exception *)`.

Remarks

Requirements

ROUTINE	REQUIRED HEADER
__setusermatherr	matherr.c

Global Variables and Standard Types

3/11/2019 • 2 minutes to read • Edit Online

The Microsoft run-time library contains definitions for global variables, control flags, and standard types used by library routines. Access these variables, flags, and types by declaring them in your program or by including the appropriate header files.

See also

C Run-Time Library Reference
Global Constants

Global Variables

3/11/2019 • 2 minutes to read • Edit Online

The Microsoft C run-time library provides the following global variables or macros. Several of these global variables or macros have been deprecated in favor of more-secure functional versions, which we recommend you use instead of the global variables.

VARIABLE	DESCRIPTION
__argc, __argv, __wargv	Contains the command-line arguments.
_daylight, _dstbias, _timezone, and _tzname	Deprecated. Instead, use `_get_daylight`, `_get_dstbias`, `_get_timezone`, and `_get_tzname`. Adjusts for local time; used in some date and time functions.
errno, _doserrno, _sys_errlist, and _sys_nerr	Deprecated. Instead, use `_get_errno`, `_set_errno`, `_get_doserrno`, `_set_doserrno`, `perror` and `strerror`. Stores error codes and related information.
_environ, _wenviron	Deprecated. Instead, use `getenv_s`, `_wgetenv_s`, `_dupenv_s`, `_wdupenv_s`, `_putenv_s`, and `_wputenv_s`. Pointers to arrays of pointers to the process environment strings; initialized at startup.
_fmode	Deprecated. Instead, use `_get_fmode` or `_set_fmode`. Sets default file-translation mode.
_iob	Array of I/O control structures for the console, files, and devices.
_pctype, _pwctype, _wctype, _mbctype, _mbcasemap	Contains information used by the character-classification functions.
_pgmptr, _wpgmptr	Deprecated. Instead, use `_get_pgmptr` or `_get_wpgmptr`. Initialized at program startup to the fully-qualified or relative path of the program, the full program name, or the program name without its file name extension, depending on how the program was invoked.

See also

C Run-Time Library Reference
Global Constants
__argc, __argv, __wargv
_get_daylight
_get_dstbias
_get_timezone

_get_tzname
perror
strerror
_get_doserrno
_set_doserrno
_get_errno
_set_errno
_dupenv_s, _wdupenv_s
getenv, _wgetenv
getenv_s, _wgetenv_s
_putenv, _wputenv
_putenv_s, _wputenv_s
_get_fmode
_set_fmode

__argc, __argv, __wargv

3/11/2019 • 2 minutes to read • Edit Online

The `__argc` global variable is a count of the number of command-line arguments passed to the program. `__argv` is a pointer to an array of single-byte-character or multi-byte-character strings that contain the program arguments, and `__wargv` is a pointer to an array of wide-character strings that contain the program arguments. These global variables provide the arguments to `main` or `wmain`.

Syntax

```
extern int __argc;
extern char ** __argv;
extern wchar_t ** __wargv;
```

Remarks

In a program that uses the `main` function, `__argc` and `__argv` are initialized at program startup by using the command line that's used to start the program. The command line is parsed into individual arguments, and wildcards are expanded. The count of arguments is assigned to `__argc` and the argument strings are allocated on the heap, and a pointer to the array of arguments is assigned to `__argv`. In a program compiled to use wide characters and a `wmain` function, the arguments are parsed and wildcards are expanded as wide-character strings, and a pointer to the array of argument strings is assigned to `__wargv`.

For portable code, we recommend you use the arguments passed to `main` to get the command-line arguments in your program.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE NOT DEFINED	_UNICODE DEFINED
`__targv`	`__argv`	`__wargv`

Requirements

GLOBAL VARIABLE	REQUIRED HEADER
`__argc`, `__argv`, `__wargv`	<stdlib.h>, <cstdlib> (C++)

`__argc`, `__argv`, and `__wargv` are Microsoft extensions. For compatibility information, see Compatibility.

See also

Global Variables
main: Program Startup
Using wmain Instead of main

_daylight, _dstbias, _timezone, and _tzname

3/11/2019 • 2 minutes to read • Edit Online

`_daylight`, `_dstbias`, `_timezone`, and `_tzname` are used in some time and date routines to make local-time adjustments. These global variables have been deprecated for the more secure functional versions, which should be used in place of the global variables.

GLOBAL VARIABLE	FUNCTIONAL EQUIVALENT
`_daylight`	_get_daylight
`_dstbias`	_get_dstbias
`_timezone`	_get_timezone
`_tzname`	_get_tzname

They are declared in Time.h as follows.

Syntax

```
extern int _daylight;
extern int _dstbias;
extern long _timezone;
extern char *_tzname[2];
```

Remarks

On a call to `_ftime`, `localtime`, or `_tzset`, the values of `_daylight`, `_dstbias`, `_timezone`, and `_tzname` are determined from the value of the `TZ` environment variable. If you do not explicitly set the value of `TZ`, `_tzname[0]` and `_tzname[1]` contain the default settings of "PST" and "PDT" respectively. The time-manipulation functions (_tzset, _ftime, and localtime) attempt to set the values of `_daylight`, `_dstbias` and `_timezone` by querying the operating system for the default value of each variable. The time-zone global variable values are shown in the following table.

VARIABLE	VALUE
`_daylight`	Nonzero if daylight saving time (DST) zone is specified in `TZ` or determined from the operating system; otherwise, 0. The default value is 1.
`_dstbias`	Offset for daylight saving time.
`_timezone`	Difference in seconds between coordinated universal time and local time. The default value is 28,800.
`_tzname[0]`	Time-zone name derived from the `TZ` environment variable. The default value is "PST".

VARIABLE	VALUE
`_tzname[1]`	DST zone name derived from the `TZ` environment variable. The default value is "PDT" (Pacific daylight time).

See also

Global Variables
_get_daylight
_get_dstbias
_get_timezone
_get_tzname

errno, _doserrno, _sys_errlist, and _sys_nerr

3/11/2019 • 2 minutes to read • Edit Online

Global macros that hold error codes that are set during program execution, and string equivalents of the error codes for display.

Syntax

```
#define errno   (*_errno())
#define _doserrno   (*__doserrno())
#define _sys_errlist (__sys_errlist())
#define _sys_nerr (*__sys_nerr())
```

Remarks

Both `errno` and `_doserrno` are set to 0 by the runtime during program startup. `errno` is set on an error in a system-level call. Because `errno` holds the value for the last call that set it, this value may be changed by succeeding calls. Run-time library calls that set `errno` on an error do not clear `errno` on success. Always clear `errno` by calling `_set_errno(0)` immediately before a call that may set it, and check it immediately after the call.

On an error, `errno` is not necessarily set to the same value as the error code returned by a system call. For I/O operations, `_doserrno` stores the operating-system error-code equivalents of `errno` codes. For most non-I/O operations, the value of `_doserrno` is not set.

Each `errno` value is associated with an error message in `_sys_errlist` that can be printed by using one of the perror functions, or stored in a string by using one of the strerror or strerror_s functions. The `perror` and `strerror` functions use the `_sys_errlist` array and `_sys_nerr`—the number of elements in `_sys_errlist`—to process error information. Direct access to `_sys_errlist` and `_sys_nerr` is deprecated for code-security reasons. We recommend that you use the more secure, functional versions instead of the global macros, as shown here:

GLOBAL MACRO	FUNCTIONAL EQUIVALENTS
`_doserrno`	_get_doserrno, _set_doserrno
`errno`	_get_errno, _set_errno
`_sys_errlist`, `_sys_nerr`	strerror_s, _strerror_s, _wcserror_s, __wcserror_s

Library math routines set `errno` by calling _matherr. To handle math errors differently, write your own routine according to the `_matherr` reference description and name it `_matherr`.

All `errno` values in the following table are predefined constants in <errno.h>, and are UNIX-compatible. Only `ERANGE`, `EILSEQ`, and `EDOM` are specified in the ISO C99 standard.

CONSTANT	SYSTEM ERROR MESSAGE	VALUE
`EPERM`	Operation not permitted	1
`ENOENT`	No such file or directory	2
`ESRCH`	No such process	3
`EINTR`	Interrupted function	4
`EIO`	I/O error	5
`ENXIO`	No such device or address	6
`E2BIG`	Argument list too long	7
`ENOEXEC`	Exec format error	8
`EBADF`	Bad file number	9
`ECHILD`	No spawned processes	10
`EAGAIN`	No more processes or not enough memory or maximum nesting level reached	11
`ENOMEM`	Not enough memory	12
`EACCES`	Permission denied	13
`EFAULT`	Bad address	14
`EBUSY`	Device or resource busy	16
`EEXIST`	File exists	17
`EXDEV`	Cross-device link	18
`ENODEV`	No such device	19
`ENOTDIR`	Not a directory	20
`EISDIR`	Is a directory	21
`EINVAL`	Invalid argument	22
`ENFILE`	Too many files open in system	23

CONSTANT	SYSTEM ERROR MESSAGE	VALUE
`EMFILE`	Too many open files	24
`ENOTTY`	Inappropriate I/O control operation	25
`EFBIG`	File too large	27
`ENOSPC`	No space left on device	28
`ESPIPE`	Invalid seek	29
`EROFS`	Read-only file system	30
`EMLINK`	Too many links	31
`EPIPE`	Broken pipe	32
`EDOM`	Math argument	33
`ERANGE`	Result too large	34
`EDEADLK`	Resource deadlock would occur	36
`EDEADLOCK`	Same as EDEADLK for compatibility with older Microsoft C versions	36
`ENAMETOOLONG`	Filename too long	38
`ENOLCK`	No locks available	39
`ENOSYS`	Function not supported	40
`ENOTEMPTY`	Directory not empty	41
`EILSEQ`	Illegal byte sequence	42
`STRUNCATE`	String was truncated	80

Requirements

GLOBAL MACRO	REQUIRED HEADER	OPTIONAL HEADER
`errno`	<errno.h> or <stdlib.h>, <cerrno> or <cstdlib> (C++)	
`_doserrno`, `_sys_errlist`, `_sys_nerr`	<stdlib.h>, <cstdlib> (C++)	<errno.h>, <cerrno> (C++)

The `_doserrno`, `_sys_errlist`, and `_sys_nerr` macros are Microsoft extensions. For more compatibility information, see Compatibility.

See also

Global Variables
errno Constants
perror, _wperror
strerror, _strerror, _wcserror, __wcserror
strerror_s, _strerror_s, _wcserror_s, __wcserror_s
_get_doserrno
_set_doserrno
_get_errno
_set_errno

_environ, _wenviron

3/11/2019 • 3 minutes to read • Edit Online

The `_environ` variable is a pointer to an array of pointers to the multibyte-character strings that constitute the process environment. This global variable has been deprecated for the more secure functional versions getenv_s, _wgetenv_s and _putenv_s, _wputenv_s, which should be used in place of the global variable. `_environ` is declared in Stdlib.h.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
extern char **_environ;
```

Remarks

In a program that uses the `main` function, `_environ` is initialized at program startup according to settings taken from the operating-system environment. The environment consists of one or more entries of the form

`ENVVARNAME` `=string`

`getenv_s` and `putenv_s` use the `_environ` variable to access and modify the environment table. When `_putenv` is called to add or delete environment settings, the environment table changes size. Its location in memory may also change, depending on the program's memory requirements. The value of `_environ` is automatically adjusted accordingly.

The `_wenviron` variable, declared in Stdlib.h as:

```
extern wchar_t **_wenviron;
```

is a wide-character version of `_environ`. In a program that uses the `wmain` function, `_wenviron` is initialized at program startup according to settings taken from the operating-system environment.

In a program that uses `main`, `_wenviron` is initially **NULL** because the environment is composed of multibyte-character strings. On the first call to `_wgetenv` or `_wputenv`, a corresponding wide-character string environment is created and is pointed to by `_wenviron`.

Similarly, in a program that uses `wmain`, `_environ` is initially **NULL** because the environment is composed of wide-character strings. On the first call to `_getenv` or `_putenv`, a corresponding multibyte-character string environment is created and is pointed to by `_environ`.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, resulting in slower execution time. For example, whenever you call `_putenv`, a call to `_wputenv` is also executed automatically, so that the two environment strings correspond.

Caution

In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, these two environment versions might not correspond exactly. This is because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. Therefore, two distinct Unicode strings might map to the same multibyte string.

Polling `_environ` in a Unicode context is meaningless when /MD or `/MDd` linkage is used. For the CRT DLL, the type (wide or multibyte) of the program is unknown. Only the multibyte type is created because that is the most likely scenario.

The following pseudo-code illustrates how this can happen.

```
int i, j;
i = _wputenv( "env_var_x=string1" );  // results in the implicit call:
                                      // putenv ("env_var_z=string1")
j = _wputenv( "env_var_y=string2" );  // also results in implicit call:
                                      // putenv("env_var_z=string2")
```

In the notation used for this example, the character strings are not C string literals; rather, they are placeholders that represent Unicode environment string literals in the `_wputenv` call and multibyte environment strings in the `putenv` call. The character placeholders '`x`' and '`y`' in the two distinct Unicode environment strings do not map uniquely to characters in the current MBCS. Instead, both map to some MBCS character '`z`' that is the default result of the attempt to convert the strings.

Thus, in the multibyte environment, the value of "`env_var_z`" after the first implicit call to `putenv` would be "`string1`", but this value would be overwritten on the second implicit call to `putenv`, when the value of "`env_var_z`" is set to "`string2`". The Unicode environment (in `_wenviron`) and the multibyte environment (in `_environ`) would therefore differ following this series of calls.

See also

Global Variables
getenv, _wgetenv
getenv_s, _wgetenv_s
_putenv, _wputenv
_putenv_s, _wputenv_s

_fmode

3/11/2019 • 2 minutes to read • Edit Online

The `_fmode` variable sets the default file-translation mode for text or binary translation. This global variable has been deprecated for the more secure functional versions _get_fmode and _set_fmode, which should be used in place of the global variable. It is declared in Stdlib.h as follows.

Syntax

```
extern int _fmode;
```

Remarks

The default setting of `_fmode` is `_O_TEXT` for text-mode translation. `_O_BINARY` is the setting for binary mode.

You can change the value of `_fmode` in three ways:

- Link with Binmode.obj. This changes the initial setting of `_fmode` to `_O_BINARY`, causing all files except `stdin`, `stdout`, and `stderr` to be opened in binary mode.
- Make a call to `_get_fmode` or `_set_fmode` to get or set the `_fmode` global variable, respectively.
- Change the value of `_fmode` directly by setting it in your program.

See also

Global Variables
_get_fmode
_set_fmode

_iob

3/11/2019 • 2 minutes to read • Edit Online

The array of stdio control structures.

Syntax

```
FILE _iob[_IOB_ENTRIES];
```

Remarks

`IOB_ENTRIES` is defined as 20 in stdio.h.

See also

Global Variables

_pctype, _pwctype, _wctype, _mbctype, _mbcasemap

3/11/2019 • 2 minutes to read • Edit Online

These global variables contain information used by the character classification functions. They are for internal use only.

Syntax

```
extern const unsigned short *_pctype;
extern const wctype_t *_pwctype;
extern const unsigned short _wctype[];
extern unsigned char _mbctype[];
extern unsigned char _mbcasemap[];
```

Remarks

The information in `_pctype`, `_pwctype`, and `_wctype` is used internally by isupper, _isupper_l, iswupper, _iswupper_l, islower, iswlower, _islower_l, _iswlower_l, isdigit, iswdigit, _isdigit_l, _iswdigit_l, isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l, isspace, iswspace, _isspace_l, _iswspace_l, isalnum, iswalnum, _isalnum_l, _iswalnum_l, ispunct, iswpunct, _ispunct_l, _iswpunct_l, isgraph, iswgraph, _isgraph_l, _iswgraph_l, iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l, toupper, _toupper, towupper, _toupper_l, _towupper_l, tolower, _tolower, towlower, _tolower_l, and _towlower_l functions. These functions should be used instead of accessing these global variables.

The information in `_mbctype` and `_mbcasemap` is used internally by _ismbbkalnum, _ismbbkalnum_l, _ismbbkana, _ismbbkana_l, _ismbbkpunct, _ismbbkpunct_l, _ismbbkprint, _ismbbkprint_l, _ismbbalpha, _ismbbpunct, _ismbbpunct_l, _ismbbalnum, _ismbbalnum_l, _ismbbprint, _ismbbprint_l, _ismbbgraph, _ismbbgraph_l, _ismbblead, _ismbblead_l, _ismbbtrail, _ismbbtrail_l, _ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l, _ismbslead, _ismbstrail, _ismbslead_l, and _ismbstrail_l. Use these functions instead of accessing the global variables.

Requirements

Not for public use.

See also

is, isw Routines
__pctype_func

_pgmptr, _wpgmptr

3/11/2019 • 2 minutes to read • Edit Online

The path of the executable file. Deprecated; use _get_pgmptr and _get_wpgmptr.

Syntax

```
extern char *_pgmptr;
extern wchar_t *_wpgmptr;
```

Remarks

When a program is run from the command interpreter (Cmd.exe), `_pgmptr` is automatically initialized to the full path of the executable file. For example, if Hello.exe is in C:\BIN and C:\BIN is in the path, `_pgmptr` is set to C:\BIN\Hello.exe when you execute:

```
C> hello
```

When a program is not run from the command line, `_pgmptr` might be initialized to the program name (the file's base name without the file name extension) or to a file name, relative path, or full path.

`_wpgmptr` is the wide-character counterpart of `_pgmptr` for use with programs that use `wmain`.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tpgmptr`	`_pgmptr`	`_pgmptr`	`_wpgmptr`

Requirements

VARIABLE	REQUIRED HEADER
`_pgmptr`, `_wpgmptr`	<stdlib.h>

Example

The following program demonstrates the use of `_pgmptr`.

```
// crt_pgmptr.c
// compile with: /W3
// The following program demonstrates the use of _pgmptr.
//
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
   printf("The full path of the executing program is : %Fs\n",
     _pgmptr); // C4996
   // Note: _pgmptr is deprecated; use _get_pgmptr instead
}
```

You could use `_wpgmptr` by changing `%Fs` to `%S` and `main` to `wmain`.

See also

Global Variables

Control Flags

3/11/2019 • 2 minutes to read • Edit Online

The debug version of the Microsoft C run-time library uses the following flags to control the heap allocation and reporting process. For more information, see CRT Debugging Techniques.

FLAG	DESCRIPTION
_CRTDBG_MAP_ALLOC	Maps the base heap functions to their debug version counterparts
_DEBUG	Enables the use of the debugging versions of the run-time functions
_crtDbgFlag	Controls how the debug heap manager tracks allocations

These flags can be defined with a /D command-line option or with a `#define` directive. When the flag is defined with `#define`, the directive must appear before the header file include statement for the routine declarations.

See also

Global Variables and Standard Types

_CRTDBG_MAP_ALLOC

3/11/2019 • 2 minutes to read • Edit Online

When the **_CRTDBG_MAP_ALLOC** flag is defined in the debug version of an application, the base version of the heap functions are directly mapped to their debug versions. The flag is used in Crtdbg.h to do the mapping. This flag is only available when the _DEBUG flag has been defined in the application.

For more information about using the debug version versus the base version of a heap function, see Using the Debug Version Versus the Base Version.

See also

Control Flags

_DEBUG

3/11/2019 • 2 minutes to read • Edit Online

The compiler defines `_DEBUG` when you specify the /MTd or /MDd option. These options specify debug versions of the C run-time library.

For more information, see CRT Debugging Techniques.

See also

Control Flags

_crtDbgFlag

3/11/2019 • 2 minutes to read • Edit Online

The **_crtDbgFlag** flag consists of five bit fields that control how memory allocations on the debug version of the heap are tracked, verified, reported, and dumped. The bit fields of the flag are set using the _CrtSetDbgFlag function. This flag and its bit fields are declared in Crtdbg.h. This flag is only available when the _DEBUG flag has been defined in the application.

For more information about using this flag in conjunction with other debug functions, see Heap State Reporting Functions.

See also

Control Flags

Standard Types

3/11/2019 • 6 minutes to read • Edit Online

The Microsoft run-time library defines the following standard types and typedefs.

Fixed-width integral types (stdint.h)

NAME	EQUIVALENT BUILT-IN TYPE
int8_t, uint8_t	signed char, unsigned char
int16_t, uint16_t	short, unsigned short
int32_t, uint32_t	int, unsigned int
int64_t, uint64_t	long long, unsigned long long
int_least8_t, uint_least8_t	signed char, unsigned char
int_least16_t, uint_least16_t	short, unsigned short
int_least32_t, uint_least32_t	int, unsigned int
int_least64_t, uint_least64_t	long long, unsigned long long
int_fast8_t, uint_fast8_t	signed char, unsigned char
int_fast16_t, uint_fast16_t	int, unsigned int
int_fast32_t, uint_fast32_t	int, unsigned int
int_fast64_t, uint_fast64_t	long long, unsigned long long
intmax_t, uintmax_t	long long, unsigned long long

TYPE	DESCRIPTION	DECLARED IN
`clock_t` (long)	Stores time values; used by clock.	TIME.H
`_complex` structure	Stores real and imaginary parts of complex numbers; used by _cabs.	MATH.H
`_CRT_ALLOC_HOOK`	A type define for the user-defined hook function. Used in _CrtSetAllocHook.	CRTDBG.H
`_CRT_DUMP_CLIENT`, `_CRT_DUMP_CLIENT_M`	A type define for a call-back function that will get called in _CrtMemDumpAllObjectsSince.	CRTDBG.H

TYPE	DESCRIPTION	DECLARED IN
`_CrtMemState` structure	Provides information about the current state of the C run-time debug heap.	CRTDBG.H
`_CRT_REPORT_HOOK`, `_CRT_REPORT_HOOKW`, `_CRT_REPORT_HOOKW_M`	A type define for a call-back function that will get called in _CrtDbgReport. The parameters for this function are: report type, output message and the return value from the call-back function.	CRTDBG.H
`dev_t`, `_dev_t` short or unsigned integer	Represents device handles.	SYS\TYPES.H
`_diskfree_t` structure	Contains information about a disk drive. Used by _getdiskfree.	DOS.H and DIRECT.H
`div_t`, `ldiv_t` and `lldiv_t` structures	Store values returned by div, ldiv, and lldiv, respectively.	STDLIB.H
`errno_t` integer	Used for a function return type or parameter that deals with the error codes of `errno`.	STDDEF.H, CRTDEFS.H
`_exception` structure	Stores error information for _matherr.	MATH.H
`_EXCEPTION_POINTERS`	Contains an exception record. See EXCEPTION_POINTERS for more information.	FPIEEE.H
`FILE` structure	Stores information about current state of stream; used in all stream I/O operations.	STDIO.H
`_finddata_t`, `_wfinddata_t`, `_finddata32_t`, `_wfinddata32_t`, `_finddatai64_t`, `_wfinddatai64_t`, `__finddata64_t`, `__wfinddata64_t`, `__finddata32i64_t`, `__wfinddata32i64_t`, `__finddata64i32_t`, `__wfinddata64i32_t` structures	Store file-attribute information returned by _findfirst, _wfindfirst, and related functions and _findnext, _wfindnext and related functions. See Filename Search Functions for information on structure members.	IO.H, WCHAR.H
`_FPIEEE_RECORD` structure	Contains information pertaining to IEEE floating-point exception; passed to user-defined trap handler by _fpieee_flt.	FPIEEE.H
`fpos_t` (long integer, `__int64`, or structure, depending on the target platform)	Used by fgetpos and fsetpos to record information for uniquely specifying every position within a file.	STDIO.H

TYPE	DESCRIPTION	DECLARED IN
`_fsize_t` (unsigned long integer)	Used to represent the size of a file.	IO.H, WCHAR.H
`_HEAPINFO` structure	Contains information about next heap entry for _heapwalk.	MALLOC.H
`_HFILE` (void *)	An operating system file handle.	CRTDBG.H
`imaxdiv_t`	The type of value that's returned by the imaxdiv function, containing both the quotient and the remainder.	inttypes.h
`ino_t`, `_ino_t` (unsigned short)	For returning status information.	WCHAR.H
`intmax_t`	A signed integer type capable of representing any value of any signed integer type.	stdint.h
`intptr_t` (long integer or `__int64`, depending on the target platform)	Stores a pointer (or HANDLE) on both Win32 and Win64 platforms.	STDDEF.H and other include files
`jmp_buf` array	Used by setjmp and longjmp to save and restore program environment.	SETJMP.H
`lconv` structure	Contains formatting rules for numeric values in different countries/regions. Used by localeconv.	LOCALE.H
`_LDOUBLE`, `_LONGDOUBLE`, `_LDBL12` (long double or an unsigned char array)	Use to represent a long double value.	STDLIB.H
`_locale_t` structure	Stores current locale values; used in all locale specific C run-time libraries.	CRTDEF.H
`mbstate_t`	Tracks the state of a multibyte character conversion.	WCHAR.H
`off_t`, `_off_t` long integer	Represents file-offset value.	WCHAR.H, SYS\TYPES.H
`_onexit_t`, `_onexit_m_t` pointer	Returned by _onexit, _onexit_m.	STDLIB.H
`_PNH` pointer to function	Type of argument to _set_new_handler.	NEW.H

TYPE	DESCRIPTION	DECLARED IN
`ptrdiff_t` (long integer or `__int64`, depending on the target platform)	Result of subtraction of two pointers.	CRTDEFS.H
`_purecall_handler`, `_purecall_handler_m`	A type define for a call-back function that is called when a pure virtual function is called. Used by _get_purecall_handler, _set_purecall_handler. A `_purecall_handler` function should have a void return type.	STDLIB.H
`_RTC_error_fn` type define	A type define for a function that will handle run-time error checks. Used in _RTC_SetErrorFunc.	RTCAPI.H
`_RTC_error_fnW` type define	A type define for a function that will handle run-time error checks. Used in _RTC_SetErrorFuncW.	RTCAPI.H
`_RTC_ErrorNumber` enumeration	Defines error conditions for _RTC_GetErrDesc and _RTC_SetErrorType.	RTCAPI.H
`_se_translator_function`	A type define for a call-back function that translates an exception. The first parameter is the exception code and the second parameter is the exception record. Used by _set_se_translator.	EH.H
`sig_atomic_t` integer	Type of object that can be modified as atomic entity, even in presence of asynchronous interrupts; used with signal.	SIGNAL.H
`size_t` (unsigned __int64 or unsigned integer, depending on the target platform)	Result of `sizeof` operator.	CRTDEFS.H and other include files
`_stat` structure	Contains file-status information returned by _stat and _fstat.	SYS\STAT.H
`__stat64` structure	Contains file-status information returned by _fstat64 and _stat64, and _wstat64.	SYS\STAT.H
`_stati64` structure	Contains file-status information returned by _fstati64, _stati64, and _wstati64.	SYS\STAT.H
`terminate_function` type define	A type define for a call-back function that is called when terminate is called. Used by set_terminate.	EH.H

TYPE	DESCRIPTION	DECLARED IN
`time_t` (__int64 or long integer)	Represents time values in mktime, time, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64 and gmtime, _gmtime32, _gmtime64. The number of seconds since January 1, 1970, 0:00 UTC. If _USE_32BIT_TIME_T is defined, `time_t` is a long integer. If not defined, it is a 64-bit integer.	TIME.H, SYS\STAT.H, SYS\TIMEB.H
`__time32_t` (long integer)	Represents time values in mktime, _mktime32, _mktime64, ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, gmtime, _gmtime32, _gmtime64 and localtime, _localtime32, _localtime64.	CRTDEFS.H, SYS\STAT.H, SYS\TIMEB.H
`__time64_t` (`__int64`)	Represents time values in mktime, _mktime32, _mktime64, _ctime64, _wctime64, ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s, _gmtime64, _localtime64 and _time64.	TIME.H, SYS\STAT.H, SYS\TIMEB.H
`_timeb` structure	Used by _ftime and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
`__timeb32` structure	Used by _ftime, _ftime32, _ftime64 and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
`__timeb64` structure	Used by _ftime64 and _ftime_s, _ftime32_s, _ftime64_s to store current system time.	SYS\TIMEB.H
`tm` structure	Used by asctime, _wasctime, asctime_s, _wasctime_s, gmtime, _gmtime32, _gmtime64, gmtime_s, _gmtime32_s, _gmtime64_s, localtime, _localtime32, _localtime64, localtime_s, _localtime32_s, _localtime64_s, mktime, _mktime32, _mktime64 and strftime, wcsftime, _strftime_l, _wcsftime_l to store and retrieve time information.	TIME.H
`uintmax_t`	An unsigned integer type capable of representing any value of any unsigned integer type.	stdint.h

TYPE	DESCRIPTION	DECLARED IN
`uintptr_t` (long integer or `__int64`, depending on the target platform)	An unsigned integer or unsigned __int64 version of `intptr_t`.	STDDEF.H and other include files
`unexpected_function`	A type define for a call-back function that is called when unexpected is called. Used by set_unexpected.	EH.H
`_utimbuf` structure	Stores file access and modification times used by _utime, _wutime and _futime, _futime32, _futime64 to change file-modification dates.	SYS\UTIME.H
`_utimbuf32` structure	Stores file access and modification times used by _utime, _utime32, _utime64, _wutime, _wutime32, _wutime64 and _futime, _futime32, _futime64 to change file-modification dates.	SYS\UTIME.H
`__utimbuf64` structure	Used by _utime64, _wutime64 and _futime64 to store the current time.	SYS\UTIME.H
`va_list` structure	Used to hold information needed by va_arg and va_end macros. Called function declares variable of type `va_list` that can be passed as argument to another function.	STDARG.H, CRTDEFS.H
`wchar_t` wide character	Useful for writing portable programs for international markets.	STDDEF.H, STDLIB.H, CRTDEFS.H, SYS\STAT.H
`wctrans_t` integer	Represents locale-specific character mappings.	WCTYPE.H
`wctype_t` integer	Can represent all characters of any language character set.	WCHAR.H, CRTDEFS.H
`wint_t` integer	Type of data object that can hold any wide character or wide end-of-file value.	WCHAR.H, CRTDEFS.H

See also

C Run-Time Library Reference

Global Constants

3/11/2019 • 2 minutes to read • Edit Online

The Microsoft run-time library contains definitions for global constants used by library routines. To use these constants, include the appropriate header files as indicated in the description for each constant. The global constants are listed in the following table.

32-Bit Windows Time/Date Formats	BUFSIZ
CLOCKS_PER_SEC, CLK_TCK	Commit-To-Disk Constants
_CRT_DISABLE_PERFCRIT_LOCKS	Data Type Constants
Environmental Constants	EOF, WEOF
errno Constants	Exception-Handling Constants
EXIT_SUCCESS, EXIT_FAILURE	File Attribute Constants
File Constants	File Permission Constants
File Read/Write Access Constants	File Translation Constants
FILENAME_MAX	FOPEN_MAX, _SYS_OPEN
_FREEENTRY, _USEDENTRY	fseek, _lseek Constants
Heap Constants	_HEAP_MAXREQ
HUGE_VAL, _HUGE	Locale Categories
_locking Constants	Math Constants
Math Error Constants	_MAX_ENV
MB_CUR_MAX	NULL
Path Field Limits	RAND_MAX
setvbuf Constants	Sharing Constants
signal Constants	signal Action Constants
spawn Constants	_stat Structure st_mode Field Constants
stdin, stdout, stderr	TMP_MAX, L_tmpnam

Translation Mode Constants	_TRUNCATE
TZNAME_MAX	_WAIT_CHILD, _WAIT_GRANDCHILD
WCHAR_MAX	WCHAR_MIN

See also

C Run-Time Library Reference
Global Variables
Considerations for Writing Prolog/Epilog Code

32-Bit Windows Time/Date Formats

3/11/2019 • 2 minutes to read • Edit Online

The file time and the date are stored individually, using unsigned integers as bit fields. File time and date are packed as follows:

Time

BIT POSITION:	0 1 2 3 4	5 6 7 8 9 A	B C D E F
Length:	5	6	5
Contents:	hours	minutes	2-second increments
Value Range:	0-23	0-59	0-29 in 2-second intervals

Date

BIT POSITION:	0 1 2 3 4 5 6	7 8 9 A	B C D E F
Length:	7	4	5
Contents:	year	month	day
Value Range:	0-119	1-12	1-31
	(relative to 1980)		

See also

Global Constants

BUFSIZ

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

`BUFSIZ` is the required user-allocated buffer for the setvbuf routine.

See also

Stream I/O
Global Constants

CLOCKS_PER_SEC, CLK_TCK

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <time.h>
```

Remarks

The time in seconds is the value returned by the `clock` function, divided by `CLOCKS_PER_SEC`. `CLK_TCK` is equivalent, but considered obsolete.

See also

clock
Global Constants

Commit-To-Disk Constants

3/11/2019 • 2 minutes to read • Edit Online

Microsoft Specific

Syntax

```
#include <stdio.h>
```

Remarks

These Microsoft-specific constants specify whether the buffer associated with the open file is flushed to operating system buffers or to disk. The mode is included in the string specifying the type of read/write access (**"r"**, **"w"**, **"a"**, **"r+"**, **"w+"**, **"a+"**).

The commit-to-disk modes are as follows:

- **c**

  Writes the unwritten contents of the specified buffer to disk. This commit-to-disk functionality only occurs at explicit calls to either the fflush or the _flushall function. This mode is useful when dealing with sensitive data. For example, if your program terminates after a call to `fflush` or `_flushall`, you can be sure that your data reached the operating system's buffers. However, unless a file is opened with the **c** option, the data might never make it to disk if the operating system also terminates.

- **n**

  Writes the unwritten contents of the specified buffer to the operating system's buffers. The operating system can cache data and then determine an optimal time to write to disk. Under many conditions, this behavior makes for efficient program behavior. However, if the retention of data is critical (such as bank transactions or airline ticket information) consider using the **c** option. The **n** mode is the default.

> **NOTE**
> The **c** and **n** options are not part of the ANSI standard for `fopen`, but are Microsoft extensions and should not be used where ANSI portability is desired.

Using the Commit-to-Disk Feature with Existing Code

By default, calls to the fflush or _flushall library functions write data to buffers maintained by the operating system. The operating system determines the optimal time to actually write the data to disk. The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating system's buffers. You can give this capability to an existing program without rewriting it by linking its object files with COMMODE.OBJ.

In the resulting executable file, calls to `fflush` write the contents of the buffer directly to disk, and calls to `_flushall` write the contents of all buffers to disk. These two functions are the only ones affected by COMMODE.OBJ.

END Microsoft Specific

See also

Stream I/O
_fdopen, _wfdopen
fopen, _wfopen
Global Constants

_CRT_DISABLE_PERFCRIT_LOCKS

3/11/2019 • 2 minutes to read • Edit Online

Disables performance-critical locking in I/O operations.

Syntax

```
#define _CRT_DISABLE_PERFCRIT_LOCKS
```

Remarks

Defining this symbol can improve performance in single-threaded I/O-bound programs by forcing all I/O operations to assume a single-threaded I/O model. For more information, see Multithreaded Libraries Performance.

See also

Global Constants

Data Type Constants

10/31/2018 • 3 minutes to read • Edit Online

Data type constants are implementation-dependent ranges of values allowed for integral and floating-point data types.

Integral type constants

These constants give the ranges for the integral data types. To use these constants, include the limits.h header in your source file:

```
#include <limits.h>
```

NOTE

The /J compiler option changes the default **char** type to **unsigned**.

CONSTANT	VALUE	DESCRIPTION
CHAR_BIT	8	Number of bits in a **char**
SCHAR_MIN	(-128)	Minimum signed **char** value
SCHAR_MAX	127	Maximum signed **char** value
UCHAR_MAX	255 (0xff)	Maximum **unsigned char** value
CHAR_MIN	(-128) (0 if **/J** option used)	Minimum **char** value
CHAR_MAX	127 (255 if **/J** option used)	Maximum **char** value
MB_LEN_MAX	5	Maximum number of bytes in multibyte **char**
SHRT_MIN	-32768	Minimum signed **short** value
SHRT_MAX	32767	Maximum signed **short** value
USHRT_MAX	65535 (0xffff)	Maximum **unsigned short** value
INT_MIN	(-2147483647 - 1)	Minimum signed **int** value
INT_MAX	2147483647	Maximum signed **int** value
UINT_MAX	4294967295 (0xffffffff)	Maximum **unsigned int** value
LONG_MIN	(-2147483647L - 1)	Minimum signed **long** value

CONSTANT	VALUE	DESCRIPTION
LONG_MAX	2147483647L	Maximum signed **long** value
ULONG_MAX	4294967295UL (0xfffffffful)	Maximum **unsigned long** value
LLONG_MIN	(-9223372036854775807LL - 1)	Minimum signed **long long** or **__int64** value
LLONG_MAX	9223372036854775807LL	Maximum signed **long long** or **__int64** value
ULLONG_MAX	0xffffffffffffffffull	Maximum **unsigned long long** value
_I8_MIN	(-127i8 - 1)	Minimum signed 8-bit value
_I8_MAX	127i8	Maximum signed 8-bit value
_UI8_MAX	0xffui8	Maximum unsigned 8-bit value
_I16_MIN	(-32767i16 - 1)	Minimum signed 16-bit value
_I16_MAX	32767i16	Maximum signed 16-bit value
_UI16_MAX	0xffffui16	Maximum unsigned 16-bit value
_I32_MIN	(-2147483647i32 - 1)	Minimum signed 32-bit value
_I32_MAX	2147483647i32	Maximum signed 32-bit value
_UI32_MAX	0xffffffffui32	Maximum unsigned 32-bit value
_I64_MIN	(-9223372036854775807 - 1)	Minimum signed 64-bit value
_I64_MAX	9223372036854775807	Maximum signed 64-bit value
_UI64_MAX	0xffffffffffffffffui64	Maximum unsigned 64-bit value
_I128_MIN	(-170141183460469231731687303715884105727i128 - 1)	Minimum signed 128-bit value
_I128_MAX	170141183460469231731687303715884105727i128	Maximum signed 128-bit value
_UI128_MAX	0xffffffffffffffffffffffffffffffffui128	Maximum unsigned 128-bit value
SIZE_MAX	same as **_UI64_MAX** if **_WIN64** is defined, or **UINT_MAX**	Maximum native integer size
RSIZE_MAX	same as (**SIZE_MAX** >> 1)	Maximum secure library integer size

Floating-point type constants

The following constants give the range and other characteristics of the **long double**, **double** and **float** data types. To use these constants, include the float.h header in your source file:

```
#include <float.h>
```

CONSTANT	VALUE	DESCRIPTION
DBL_DECIMAL_DIG	17	# of decimal digits of rounding precision
DBL_DIG	15	# of decimal digits of precision
DBL_EPSILON	2.2204460492503131e-016	Smallest such that 1.0 + **DBL_EPSILON** != 1.0
DBL_HAS_SUBNORM	1	Type supports subnormal (denormal) numbers
DBL_MANT_DIG	53	# of bits in significand (mantissa)
DBL_MAX	1.7976931348623158e+308	Maximum value
DBL_MAX_10_EXP	308	Maximum decimal exponent
DBL_MAX_EXP	1024	Maximum binary exponent
DBL_MIN	2.2250738585072014e-308	Minimum normalized positive value
DBL_MIN_10_EXP	(-307)	Minimum decimal exponent
DBL_MIN_EXP	(-1021)	Minimum binary exponent
_DBL_RADIX	2	Exponent radix
DBL_TRUE_MIN	4.9406564584124654e-324	Minimum positive subnormal value
FLT_DECIMAL_DIG	9	Number of decimal digits of rounding precision
FLT_DIG	6	Number of decimal digits of precision
FLT_EPSILON	1.192092896e-07F	Smallest such that 1.0 + **FLT_EPSILON** != 1.0
FLT_HAS_SUBNORM	1	Type supports subnormal (denormal) numbers
FLT_MANT_DIG	24	Number of bits in significand (mantissa)
FLT_MAX	3.402823466e+38F	Maximum value
FLT_MAX_10_EXP	38	Maximum decimal exponent
FLT_MAX_EXP	128	Maximum binary exponent

CONSTANT	VALUE	DESCRIPTION
FLT_MIN	1.175494351e-38F	Minimum normalized positive value
FLT_MIN_10_EXP	(-37)	Minimum decimal exponent
FLT_MIN_EXP	(-125)	Minimum binary exponent
FLT_RADIX	2	Exponent radix
FLT_TRUE_MIN	1.401298464e-45F	Minimum positive subnormal value
LDBL_DIG	15	# of decimal digits of precision
LDBL_EPSILON	2.2204460492503131e-016	Smallest such that 1.0 + **LDBL_EPSILON** != 1.0
LDBL_HAS_SUBNORM	1	Type supports subnormal (denormal) numbers
LDBL_MANT_DIG	53	# of bits in significand (mantissa)
LDBL_MAX	1.7976931348623158e+308	Maximum value
LDBL_MAX_10_EXP	308	Maximum decimal exponent
LDBL_MAX_EXP	1024	Maximum binary exponent
LDBL_MIN	2.2250738585072014e-308	Minimum normalized positive value
LDBL_MIN_10_EXP	(-307)	Minimum decimal exponent
LDBL_MIN_EXP	(-1021)	Minimum binary exponent
_LDBL_RADIX	2	Exponent radix
LDBL_TRUE_MIN	4.9406564584124654e-324	Minimum positive subnormal value
DECIMAL_DIG	same as **DBL_DECIMAL_DIG**	Default (double) decimal digits of rounding precision

See also

Global Constants

Environmental Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdlib.h>
```

Remarks

This constant defines the environmental length for strings.

CONSTANT	MEANING
`_MAX_ENV`	Maximum string size of an environmental string.

See also

Global Constants

EOF, WEOF

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

EOF is returned by an I/O routine when the end-of-file (or in some cases, an error) is encountered.

WEOF yields the return value, of type **wint_t**, used to signal the end of a wide stream, or to report an error condition.

See also

putc, putwc
ungetc, ungetwc
scanf, _scanf_l, wscanf, _wscanf_l
fflush
fclose, _fcloseall
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock
_putch, _putwch
isascii, __isascii, iswascii
Global Constants

errno Constants

3/11/2019 • 6 minutes to read • Edit Online

Syntax

```
#include <errno.h>
```

Remarks

The **errno** values are constants assigned to errno in the event of various error conditions.

ERRNO.H contains the definitions of the **errno** values. However, not all the definitions given in ERRNO.H are used in 32-bit Windows operating systems. Some of the values in ERRNO.H are present to maintain compatibility with the UNIX family of operating systems.

The **errno** values in a 32-bit Windows operating system are a subset of the values for **errno** in XENIX systems. Thus, the **errno** value is not necessarily the same as the actual error code returned by a system call from the Windows operating systems. To access the actual operating system error code, use the _doserrno variable, which contains this value.

The following **errno** values are supported:

CONSTANT	DESCRIPTION
ECHILD	No spawned processes.
EAGAIN	No more processes. An attempt to create a new process failed because there are no more process slots, or there is not enough memory, or the maximum nesting level has been reached.
E2BIG	Argument list too long.
EACCES	Permission denied. The file's permission setting does not allow the specified access. This error signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes. For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under MS-DOS operating system versions 3.0 and later, **EACCES** may also indicate a locking or sharing violation. The error can also occur in an attempt to rename a file or directory or to remove an existing directory.
EBADF	Bad file number. There are two possible causes: 1) The specified file descriptor is not a valid value or does not refer to an open file. 2) An attempt was made to write to a file or device opened for read-only access.

CONSTANT	DESCRIPTION
EDEADLOCK	Resource deadlock would occur. The argument to a math function is not in the domain of the function.
EDOM	Math argument.
EEXIST	Files exist. An attempt has been made to create a file that already exists. For example, the **_O_CREAT** and **_O_EXCL** flags are specified in an **_open** call, but the named file already exists.
EILSEQ	Illegal sequence of bytes (for example, in an MBCS string).
EINVAL	Invalid argument. An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer (by means of a call to **fseek**) is before the beginning of the file.
EMFILE	Too many open files. No more file descriptors are available, so no more files can be opened.
ENOENT	No such file or directory. The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path does not specify an existing directory.
ENOEXEC	Exec format error. An attempt was made to execute a file that is not executable or that has an invalid executable-file format.
ENOMEM	Not enough core. Not enough memory is available for the attempted operator. For example, this message can occur when insufficient memory is available to execute a child process, or when the allocation request in a **_getcwd** call cannot be satisfied.
ENOSPC	No space left on device. No more space for writing is available on the device (for example, when the disk is full).
ERANGE	Result too large. An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the *buffer* argument to **_getcwd** is longer than expected).
EXDEV	Cross-device link. An attempt was made to move a file to a different device (using the **rename** function).
STRUNCATE	A string copy or concatenation resulted in a truncated string. See _TRUNCATE.

The following values are supported for compatibility with Posix. They are required values on non-Posix systems.

```
#define E2BIG /* argument list too long */
#define EACCES /* permission denied */
#define EADDRINUSE /* address in use */
#define EADDRNOTAVAIL /* address not available */
#define EAFNOSUPPORT /* address family not supported */
```

```
#define EAGAIN /* resource unavailable try again */
#define EALREADY /* connection already in progress */
#define EBADF /* bad file descriptor */
#define EBADMSG /* bad message */
#define EBUSY /* device or resource busy */
#define ECANCELED /* operation canceled */
#define ECHILD /* no child process */
#define ECONNABORTED /* connection aborted */
#define ECONNREFUSED /* connection refused */
#define ECONNRESET /* connection reset */
#define EDEADLK /* resource deadlock would occur */
#define EDESTADDRREQ /* destination address required */
#define EDOM /* argument out of domain */
#define EEXIST /* file exists */
#define EFAULT /* bad address */
#define EFBIG /* file too large */
#define EHOSTUNREACH /* host unreachable */
#define EIDRM /* identifier removed */
#define EILSEQ /* illegal byte sequence */
#define EINPROGRESS /* operation in progress */
#define EINTR /* interrupted */
#define EINVAL /* invalid argument */
#define EIO /* io error */
#define EISCONN /* already connected */
#define EISDIR /* is a directory */
#define ELOOP /* too many synbolic link levels */
#define EMFILE /* too many files open */
#define EMLINK /* too many links */
#define EMSGSIZE /* message size */
#define ENAMETOOLONG /* filename too long */
#define ENETDOWN /* network down */
#define ENETRESET /* network reset */
#define ENETUNREACH /* network unreachable */
#define ENFILE /* too many files open in system */
#define ENOBUFS /* no buffer space */
#define ENODATA /* no message available */
#define ENODEV /* no such device */
#define ENOENT /* no such file or directory */
#define ENOEXEC /* executable format error */
#define ENOLCK /* no lock available */
#define ENOLINK /* no link */
#define ENOMEM /* not enough memory */
#define ENOMSG /* no message */
#define ENOPROTOOPT /* no protocol option */
#define ENOSPC /* no space on device */
#define ENOSR /* no stream resources */
#define ENOSTR /* not a stream */
#define ENOSYS /* function not supported */
#define ENOTCONN /* not connected */
#define ENOTDIR /* not a directory */
#define ENOTEMPTY /* directory not empty */
#define ENOTRECOVERABLE /* state not recoverable */
#define ENOTSOCK /* not a socket */
#define ENOTSUP /* not supported */
#define ENOTTY /* inappropriate io control operation */
#define ENXIO /* no such device or address */
#define EOPNOTSUPP /* operation not supported */
#define EOTHER /* other */
#define EOVERFLOW /* value too large */
#define EOWNERDEAD /* owner dead */
#define EPERM /* operation not permitted */
#define EPIPE /* broken pipe */
#define EPROTO /* protocol error */
#define EPROTONOSUPPORT /* protocol not supported */
#define EPROTOTYPE /* wrong protocol type */
#define ERANGE /* result out of range */
#define EROFS /* read only file system */
#define ESPIPE /* invalid seek */
#define ESRCH /* no such process */
```

```
#define ETIME /* stream timeout */
#define ETIMEDOUT /* timed out */
#define ETXTBSY /* text file busy */
#define EWOULDBLOCK /* operation would block */
#define EXDEV /* cross device link */
```

See also

Global Constants

Exception-Handling Constants

3/11/2019 • 2 minutes to read • Edit Online

The constant `EXCEPTION_CONTINUE_SEARCH`, `EXCEPTION_CONTINUE_EXECUTION`, or `EXCEPTION_EXECUTE_HANDLER` is returned when an exception occurs during execution of the guarded section of a **try-except** statement. The return value determines how the exception is handled. For more information, see try-except Statement in the *C++ Language Reference*.

See also

Global Constants

EXIT_SUCCESS, EXIT_FAILURE

10/31/2018 • 2 minutes to read • Edit Online

Required header

```
#include <stdlib.h>
```

Remarks

These are arguments for the exit and _exit functions, and the return values for the atexit and _onexit functions.

CONSTANT	DEFINED VALUE
EXIT_SUCCESS	0
EXIT_FAILURE	1

See also

Global Constants

File Attribute Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <io.h>
```

Remarks

These constants specify the current attributes of the file or directory specified by the function.

The attributes are represented by the following manifest constants:

CONSTANT	DESCRIPTION
`_A_ARCH`	Archive. Set whenever the file is changed, and cleared by the BACKUP command. Value: 0x20
`_A_HIDDEN`	Hidden file. Not normally seen with the DIR command, unless the /AH option is used. Returns information about normal files as well as files with this attribute. Value: 0x02
`_A_NORMAL`	Normal. File can be read or written to without restriction. Value: 0x00
`_A_RDONLY`	Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Value: 0x01
`_A_SUBDIR`	Subdirectory. Value: 0x10
`_A_SYSTEM`	System file. Not normally seen with the DIR command, unless the /AS option is used. Value: 0x04

Multiple constants can be combined with the OR operator (|).

See also

Filename Search Functions
Global Constants

File Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <fcntl.h>
```

Remarks

The integer expression formed from one or more of these constants determines the type of reading or writing operations permitted. It is formed by combining one or more constants with a translation-mode constant.

The file constants are as follows:

CONSTANT	DESCRIPTION
`_O_APPEND`	Repositions the file pointer to the end of the file before every write operation.
`_O_CREAT`	Creates and opens a new file for writing; this has no effect if the file specified by *filename* exists.
`_O_EXCL`	Returns an error value if the file specified by *filename* exists. Only applies when used with `_O_CREAT`.
`_O_RDONLY`	Opens file for reading only; if this flag is given, neither `_O_RDWR` nor `_O_WRONLY` can be given.
`_O_RDWR`	Opens file for both reading and writing; if this flag is given, neither `_O_RDONLY` nor `_O_WRONLY` can be given.
`_O_TRUNC`	Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed. If this flag is given, you cannot specify `_O_RDONLY`.
`_O_WRONLY`	Opens file for writing only; if this flag is given, neither `_O_RDONLY` nor `_O_RDWR` can be given.

See also

_open, _wopen
_sopen, _wsopen
Global Constants

File Permission Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <sys/stat.h>
```

Remarks

One of these constants is required when `_O_CREAT` (`_open` , `_sopen`) is specified.

The `pmode` argument specifies the file's permission settings as follows.

CONSTANT	MEANING
`_S_IREAD`	Reading permitted
`_S_IWRITE`	Writing permitted
`_S_IREAD` | `_S_IWRITE`	Reading and writing permitted

When used as the `pmode` argument for `_umask` , the manifest constant sets the permission setting, as follows.

CONSTANT	MEANING
`_S_IREAD`	Writing not permitted (file is read-only)
`_S_IWRITE`	Reading not permitted (file is write-only)
`_S_IREAD` | `_S_IWRITE`	Neither reading nor writing permitted

See also

_open, _wopen
_sopen, _wsopen
_umask
Standard Types
Global Constants

File Read/Write Access Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

These constants specify the access type ("a", "r", or "w") requested for the file. Both the translation mode ("b" or "t") and the commit-to-disk mode ("c" or "n") can be specified with the type of access.

The access types are described in this table:

ACCESS TYPE	DESCRIPTION
"r"	Opens for reading. If the file does not exist or cannot be found, the call to open the file fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending); creates the file first if it does not exist. All write operations occur at the end of the file. Although the file pointer can be repositioned using `fseek` or `rewind`, it is always moved back to the end of the file before any write operation is carried out.
"r+"	Opens for both reading and writing. If the file does not exist or cannot be found, the call to open the file fails.
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	The same as **"a"** but also allows reading.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening `fflush`, `fsetpos`, `fseek`, or `rewind` operation. The current position can be specified for the `fsetpos` or `fseek` operation.

See also

_fdopen, _wfdopen
fopen, _wfopen
freopen, _wfreopen
_fsopen, _wfsopen
_popen, _wpopen
Global Constants

File Translation Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

These constants specify the mode of translation (**"b"** or **"t"**). The mode is included in the string specifying the type of access (**"r"**, **"w"**, **"a"**, **"r+"**, **"w+"**, **"a+"**).

The translation modes are as follows:

- **t**

  Opens in text (translated) mode. In this mode, carriage-return/linefeed (CR-LF) combinations are translated into single linefeeds (LF) on input, and LF characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, `fopen` checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the `fseek` and `ftell` functions to move within a file ending with CTRL+Z may cause `fseek` to behave improperly near the end of the file.

  > **NOTE**
  > The **t** option is not part of the ANSI standard for `fopen` and `freopen`. It is a Microsoft extension and should not be used where ANSI portability is desired.

- **b**

  Opens in binary (untranslated) mode. The above translations are suppressed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable _fmode. For more information about using text and binary modes, see Text and Binary Mode File I/O.

See also

_fdopen, _wfdopen
fopen, _wfopen
freopen, _wfreopen
_fsopen, _wfsopen
Global Constants

FILENAME_MAX

3/11/2019 • 2 minutes to read • Edit Online

The maximum permissible length for a `filename` string buffer size.

Syntax

```
#include <stdio.h>
```

See also

Path Field Limits
Global Constants

FOPEN_MAX, _SYS_OPEN

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

This is the maximum number of files that can be opened simultaneously. `FOPEN_MAX` is the ANSI-compatible name. `_SYS_OPEN` is provided for compatibility with existing code.

See also

Global Constants

_FREEENTRY, _USEDENTRY

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <malloc.h>
```

Remarks

These constants represent values assigned by the `_heapwalk` routines to the **_useflag** element of the **_HEAPINFO** structure. They indicate the status of the heap entry.

See also

_heapwalk
Global Constants

fseek, _lseek Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

The *origin* argument specifies the initial position and can be one of the following manifest constants:

CONSTANT	MEANING
`SEEK_END`	End of file
`SEEK_CUR`	Current position of file pointer
`SEEK_SET`	Beginning of file

See also

fseek, _fseeki64
_lseek, _lseeki64
Global Constants

Heap Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <malloc.h>
```

Remarks

These constants give the return value indicating status of the heap.

CONSTANT	MEANING
`_HEAPBADBEGIN`	Initial header information was not found or was invalid.
`_HEAPBADNODE`	Bad node was found, or heap is damaged.
`_HEAPBADPTR`	**_pentry** field of **_HEAPINFO** structure does not contain valid pointer into heap (`_heapwalk` routine only).
`_HEAPEMPTY`	Heap has not been initialized.
`_HEAPEND`	End of heap was reached successfully (`_heapwalk` routine only).
`_HEAPOK`	Heap is consistent (`_heapset` and `_heapchk` routines only). No errors so far; **_HEAPINFO** structure contains information about next entry (`_heapwalk` routine only).

See also

_heapchk
_heapset
_heapwalk
Global Constants

_HEAP_MAXREQ

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <malloc.h>
```

Remarks

The maximum size of a user request for memory that can be granted.

See also

malloc
calloc
Global Constants

HUGE_VAL, _HUGE

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <math.h>
```

Remarks

`HUGE_VAL` is the largest representable double value. This value is returned by many run-time math functions when an error occurs. For some functions, -`HUGE_VAL` is returned. `HUGE_VAL` is defined as `_HUGE`, but run-time math functions return `HUGE_VAL`. You should also use `HUGE_VAL` in your code for consistency.

See also

Global Constants

Locale Categories

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <locale.h>
```

Remarks

Locale categories are manifest constants used by the localization routines to specify which portion of a program's locale information will be used. The locale refers to the locality (or Country/Region) for which certain aspects of your program can be customized. Locale-dependent areas include, for example, the formatting of dates or the display format for monetary values.

LOCALE CATEGORY	PARTS OF PROGRAM AFFECTED
`LC_ALL`	All locale-specific behavior (all categories)
`LC_COLLATE`	Behavior of `strcoll` and `strxfrm` functions
`LC_CTYPE`	Behavior of character-handling functions (except `isdigit`, `isxdigit`, `mbstowcs`, and `mbtowc`, which are unaffected)
`LC_MAX`	Same as `LC_TIME`
`LC_MIN`	Same as `LC_ALL`
`LC_MONETARY`	Monetary formatting information returned by the `localeconv` function
`LC_NUMERIC`	Decimal-point character for formatted output routines (for example, `printf`), data conversion routines, and nonmonetary formatting information returned by `localeconv` function
`LC_TIME`	Behavior of `strftime` function

See setlocale, _wsetlocale for an example.

See also

localeconv
setlocale, _wsetlocale
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
Global Constants

_locking Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <sys/locking.h>
```

Remarks

The *mode* argument in the call to the `_locking` function specifies the locking action to be performed.

The *mode* argument must be one of the following manifest constants.

`_LK_LOCK`	Locks the specified bytes. If the bytes cannot be locked, the function tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, the function returns an error.
`_LK_RLCK`	Same as `_LK_LOCK`.
`_LK_NBLCK`	Locks the specified bytes. If bytes cannot be locked, the function returns an error.
`_LK_NBRLCK`	Same as `_LK_NBLCK`.
`_LK_UNLCK`	Unlocks the specified bytes. (The bytes must have been previously locked.)

See also

_locking
Global Constants

Math Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#define _USE_MATH_DEFINES // for C++
#include <cmath>

#define _USE_MATH_DEFINES // for C
#include <math.h>
```

Remarks

The following symbols are defined for the values of their indicated expressions:

SYMBOL	EXPRESSION	VALUE
M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.3025850929940456840 2
M_PI	pi	3.14159265358979323846
M_PI_2	pi/2	1.57079632679489661923
M_PI_4	pi/4	0.785398163397448309616
M_1_PI	1/pi	0.318309886183790671538
M_2_PI	2/pi	0.636619772367581343076
M_2_SQRTPI	2/sqrt(pi)	1.12837916709551257390
M_SQRT2	sqrt(2)	1.41421356237309504880
M_SQRT1_2	1/sqrt(2)	0.707106781186547524401

Math Constants are not defined in Standard C/C++. To use them, you must first define `_USE_MATH_DEFINES` and then include cmath or math.h.

The file ATLComTime.h includes math.h when your project is built in Release mode. If you use one or more of the math constants in a project that also includes ATLComTime.h, you must define `_USE_MATH_DEFINES` before you include ATLComTime.h.

See also

Global Constants

Math Error Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <math.h>
```

Remarks

The math routines of the run-time library can generate math error constants.

These errors, described as follows, correspond to the exception types defined in MATH.H and are returned by the `_matherr` function when a math error occurs.

CONSTANT	MEANING
`_DOMAIN`	Argument to function is outside domain of function.
`_OVERFLOW`	Result is too large to be represented in function's return type.
`_PLOSS`	Partial loss of significance occurred.
`_SING`	Argument singularity: argument to function has illegal value. (For example, value 0 is passed to function that requires nonzero value.)
`_TLOSS`	Total loss of significance occurred.
`_UNDERFLOW`	Result is too small to be represented.

See also

_matherr
Global Constants

_MAX_ENV

3/11/2019 • 2 minutes to read • Edit Online

The maximum permissible string length of an environmental variable.

Syntax

```
#include <stdio.h>
```

See also

Environmental Constants
Global Constants

MB_CUR_MAX

3/11/2019 • 2 minutes to read • Edit Online

A macro that indicates the maximum number of bytes in a multibyte character for the current locale.

Syntax

```
#include <stdlib.h>
```

Remarks

Context: ANSI multibyte- and wide-character conversion functions

The value of `MB_CUR_MAX` is the maximum number of bytes in a multibyte character for the current locale.

See also

_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
___mb_cur_max_func, ___mb_cur_max_l_func, __p___mb_cur_max, __mb_cur_max
Standard Types
wcstombs, _wcstombs_l
wctomb, _wctomb_l
Data Type Constants
Global Constants

NULL (CRT)

3/11/2019 • 2 minutes to read • Edit Online

NULL is the null-pointer value used with many pointer operations and functions. It is equivalent to 0. **NULL** is defined in the following header files: CRTDBG.H, LOCALE.H, STDDEF.H, STDIO.H, STDLIB.H, STRING.H, TCHAR.H, TIME.H and WCHAR.H.

See also

Global Constants

Path Field Limits

10/31/2018 • 2 minutes to read • Edit Online

Syntax

```
#include <stdlib.h>
```

Remarks

These constants define the maximum length for the path and for the individual fields within the path.

CONSTANT	MEANING
`_MAX_DIR`	Maximum length of directory component
`_MAX_DRIVE`	Maximum length of drive component
`_MAX_EXT`	Maximum length of extension component
`_MAX_FNAME`	Maximum length of filename component
`_MAX_PATH`	Maximum length of full path

NOTE

The C Runtime supports path lengths up to 32768 characters in length, but it is up to the operating system, specifically the file system, to support these longer paths. The sum of the fields should not exceed `_MAX_PATH` for full backwards compatibility with FAT32 file systems. The Windows NTFS file system supports paths up to 32768 characters in length, but only when using the Unicode APIs. When using long path names, prefix the path with the characters \\?\ and use the Unicode versions of the C Runtime functions.

See also

Global Constants

RAND_MAX

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdlib.h>
```

Remarks

The constant `RAND_MAX` is the maximum value that can be returned by the `rand` function. `RAND_MAX` is defined as the value 0x7fff.

See also

rand
Global Constants

setvbuf Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

These constants represent the type of buffer for `setvbuf`.

The possible values are given by the following manifest constants:

CONSTANT	MEANING
`_IOFBF`	Full buffering: Buffer specified in call to `setvbuf` is used and its size is as specified in `setvbuf` call. If buffer pointer is **NULL**, automatically allocated buffer of specified size is used.
`_IOLBF`	Same as `_IOFBF`.
`_IONBF`	No buffer is used, regardless of arguments in call to `setvbuf`.

See also

setbuf
Global Constants

Sharing Constants

3/11/2019 • 2 minutes to read • Edit Online

Constants for file-sharing modes.

Syntax

```
#include <share.h>
```

Remarks

The *shflag* argument determines the sharing mode, which consists of one or more manifest constants. These can be combined with the *oflag* arguments (see File Constants).

The following table lists the constants and their meanings:

CONSTANT	MEANING
`_SH_DENYRW`	Denies read and write access to file
`_SH_DENYWR`	Denies write access to file
`_SH_DENYRD`	Denies read access to file
`_SH_DENYNO`	Permits read and write access
`_SH_SECURE`	Sets secure mode (shared read, exclusive write access).

See also

_sopen, _wsopen
_fsopen, _wfsopen
Global Constants

signal Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <signal.h>
```

Remarks

The `sig` argument must be one of the manifest constants listed below (defined in SIGNAL.H).

SIGABRT	Abnormal termination. The default action terminates the calling program with exit code 3.
SIGABRT_COMPAT	Same as SIGABRT. For compatibility with other platforms.
SIGFPE	Floating-point error, such as overflow, division by zero, or invalid operation. The default action terminates the calling program.
SIGILL	Illegal instruction. The default action terminates the calling program.
SIGINT	CTRL+C interrupt. The default action terminates the calling program with exit code 3.
SIGSEGV	Illegal storage access. The default action terminates the calling program.
SIGTERM	Termination request sent to the program. The default action terminates the calling program with exit code 3.
SIG_ERR	A return type from a signal indicating an error has occurred.

See also

signal
raise
Global Constants

signal Action Constants

3/11/2019 • 2 minutes to read • Edit Online

The action taken when the interrupt signal is received depends on the value of `func`.

Syntax

```
#include <signal.h>
```

Remarks

The `func` argument must be either a function address or one of the manifest constants listed below and defined in SIGNAL.H.

`SIG_DFL`	Uses system-default response. If the calling program uses stream I/O, buffers created by the run-time library are not flushed.
`SIG_IGN`	Ignores interrupt signal. This value should never be given for `SIGFPE`, since the floating-point state of the process is left undefined.
`SIG_SGE`	Indicates an error occurred in the signal.
`SIG_ACK`	Indicates an acknowledgement was received.
`SIG_ERR`	A return type from a signal indicating an error has occurred.

See also

signal
Global Constants

spawn Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <process.h>
```

Remarks

The `mode` argument determines the action taken by the calling process before and during a spawn operation. The following values for `mode` are possible:

CONSTANT	MEANING
`_P_OVERLAY`	Overlays calling process with new process, destroying calling process (same effect as `_exec` calls).
`_P_WAIT`	Suspends calling thread until execution of new process is complete (synchronous `_spawn`).
`_P_NOWAIT` , `_P_NOWAITO`	Continues to execute calling process concurrently with new process (asynchronous `_spawn`).
`_P_DETACH`	Continues to execute calling process; new process is run in background with no access to console or keyboard. Calls to `_cwait` against new process will fail. This is an asynchronous `_spawn` .

See also

_spawn, _wspawn Functions
Global Constants

_stat Structure st_mode Field Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <sys/stat.h>
```

Remarks

These constants are used to indicate file type in the **st_mode** field of the _stat structure.

The bit mask constants are described below:

CONSTANT	MEANING
`_S_IFMT`	File type mask
`_S_IFDIR`	Directory
`_S_IFCHR`	Character special (indicates a device if set)
`_S_IFREG`	Regular
`_S_IREAD`	Read permission, owner
`_S_IWRITE`	Write permission, owner
`_S_IEXEC`	Execute/search permission, owner

See also

_stat, _wstat Functions
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
Standard Types
Global Constants

stdin, stdout, stderr

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
#include <stdio.h>
```

Remarks

These are standard streams for input, output, and error output.

By default, standard input is read from the keyboard, while standard output and standard error are printed to the screen.

The following stream pointers are available to access the standard streams:

POINTER	STREAM
`stdin`	Standard input
`stdout`	Standard output
`stderr`	Standard error

These pointers can be used as arguments to functions. Some functions, such as getchar and putchar, use `stdin` and `stdout` automatically.

These pointers are constants, and cannot be assigned new values. The freopen function can be used to redirect the streams to disk files or to other devices. The operating system allows you to redirect a program's standard input and output at the command level.

See also

Stream I/O
Global Constants

TMP_MAX, L_tmpnam

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <stdio.h>
```

Remarks

`TMP_MAX` is the maximum number of unique filenames that the `tmpnam` function can generate. `L_tmpnam` is the length of temporary filenames generated by `tmpnam`.

See also

Global Constants

Translation Mode Constants

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <fcntl.h>
```

Remarks

The `_O_BINARY` and `_O_TEXT` manifest constants determine the translation mode for files (`_open` and `_sopen`) or the translation mode for streams (`_setmode`).

The allowed values are:

`_O_TEXT`	Opens file in text (translated) mode. Carriage return - linefeed (CR-LF) combinations are translated into a single linefeed (LF) on input. Linefeed characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading and reading/writing, `fopen` checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the `fseek` and `ftell` functions to move within a file ending with CTRL+Z may cause `fseek` to behave improperly near the end of the file.
`_O_BINARY`	Opens file in binary (untranslated) mode. The above translations are suppressed.
`_O_RAW`	Same as `_O_BINARY`. Supported for C 2.0 compatibility.

For more information, see Text and Binary Mode File I/O and File Translation.

See also

_open, _wopen
_pipe
_sopen, _wsopen
_setmode
Global Constants

_TRUNCATE

3/11/2019 • 2 minutes to read • Edit Online

Specifies string truncation behavior.

Syntax

```
#include <stdlib.h>
```

Remarks

`_TRUNCATE` enables truncation behavior when passed as the `count` parameter to these functions:

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

mbstowcs_s, _mbstowcs_s_l

mbsrtowcs_s

wcstombs_s, _wcstombs_s_l

wcsrtombs_s

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

If the destination buffer is too small to hold the entire string, the normal behavior of these functions is to treat it as an error situation (see Parameter Validation). However, if string truncation is enabled by passing `_TRUNCATE`, these functions will copy only as much of the string as will fit, leaving the destination buffer null-terminated, and return successfully.

String truncation changes the return values of the affected functions. The following functions return 0 if no truncation occurs, or `STRUNCATE` if truncation does occur:

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

wcstombs_s, _wcstombs_s_l

mbstowcs_s, _mbstowcs_s_l

The following functions return the number of characters copied if no truncation occurs, or -1 if truncation does occur (matching the behavior of the original snprintf functions):

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

Example

```
// crt_truncate.c
#include <stdlib.h>
#include <errno.h>

int main()
{
   char src[] = "1234567890";
   char dst[5];
   errno_t err = strncpy_s(dst, _countof(dst), src, _TRUNCATE);
   if ( err == STRUNCATE )
      printf( "truncation occurred!\n" );
   printf( "'%s'\n", dst );
}
```

```
truncation occurred!
'1234'
```

See also

Global Constants

TZNAME_MAX

3/11/2019 • 2 minutes to read • Edit Online

Obsolete. The maximum permissible string length for a time zone name variable. This macro was defined in <limits.h> in Visual Studio 2012 and earlier versions. It is not defined in Visual Studio 2013 and later versions. To get the length required to hold the current time zone name, use _get_tzname.

Syntax

```
#include <limits.h>
```

See also

Environmental Constants
Global Constants
_get_tzname

_WAIT_CHILD, _WAIT_GRANDCHILD

3/11/2019 • 2 minutes to read • Edit Online

Syntax

```
#include <process.h>
```

Remarks

The `_cwait` function can be used by any process to wait for any other process (if the process ID is known). The action argument can be one of the following values:

CONSTANT	MEANING
`_WAIT_CHILD`	Calling process waits until specified new process terminates.
`_WAIT_GRANDCHILD`	Calling process waits until specified new process, and all processes created by that new process, terminate.

See also

_cwait
Global Constants

WCHAR_MAX

3/11/2019 • 2 minutes to read • Edit Online

Maximum value for type `wchar_t`.

Syntax

```
#include <wchar.h>
```

See also

Global Constants

WCHAR_MIN

3/11/2019 • 2 minutes to read • Edit Online

Minimum value for type `wchar_t`.

Syntax

```
#include <wchar.h>
```

See also

Global Constants

Generic-Text Mappings

3/11/2019 • 2 minutes to read • Edit Online

To simplify writing code for international markets, generic-text mappings are defined in TCHAR.H for:

- Data types
- Constants and global variables
- Routine mappings

For more information, see Using Generic-Text Mappings. Generic-text mappings are Microsoft extensions that are not ANSI compatible.

See also

Data Type Mappings
A Sample Generic-Text Program

Data Type Mappings

3/11/2019 • 2 minutes to read • Edit Online

These data-type mappings are defined in TCHAR.H and depend on whether the constant `_UNICODE` or `_MBCS` has been defined in your program.

For related information, see Using TCHAR.H Data Types with _MBCS Code.

Generic-Text Data Type Mappings

GENERIC-TEXT DATA TYPE NAME	SBCS (_UNICODE, _MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_TCHAR`	`char`	`char`	`wchar_t`
`_tfinddata_t`	`_finddata_t`	`_finddata_t`	`_wfinddata_t`
`_tfinddata64_t`	`__finddata64_t`	`__finddata64_t`	`__wfinddata64_t`
`_tfinddatai64_t`	`_finddatai64_t`	`_finddatai64_t`	`_wfinddatai64_t`
`_TINT`	`int`	`int`	`wint_t`
`_TSCHAR`	`signed char`	`signed char`	`wchar_t`
`_TUCHAR`	`unsigned char`	`unsigned char`	`wchar_t`
`_TXCHAR`	`char`	`unsigned char`	`wchar_t`
`_T` or `_TEXT`	No effect (removed by preprocessor)	No effect (removed by preprocessor)	`L` (converts following character or string to its Unicode counterpart)

See also

Generic-Text Mappings
Constant and Global Variable Mappings
Routine Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

Constant and Global Variable Mappings

3/11/2019 • 2 minutes to read • Edit Online

These generic-text constant, global variable, and standard-type mappings are defined in TCHAR.H and depend on whether the constant `_UNICODE` or `_MBCS` has been defined in your program.

Generic-Text Constant and Global Variable Mappings

GENERIC-TEXT - OBJECT NAME	SBCS (_UNICODE, _MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_TEOF`	`EOF`	`EOF`	`WEOF`
`_tenviron`	`_environ`	`_environ`	`_wenviron`
`_tpgmptr`	`_pgmptr`	`_pgmptr`	`_wpgmptr`

See also

Generic-Text Mappings
Data Type Mappings
Routine Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

Routine Mappings

3/11/2019 • 5 minutes to read • Edit Online

The generic-text routine mappings are defined in TCHAR.H. `_tccpy` and `_tclen` map to functions in the MBCS model; they are mapped to macros or inline functions in the SBCS and Unicode models for completeness. For information on a generic text routine, see the help topic about the corresponding `SBCS` -, `_MBCS` -, or `_UNICODE` -related routine.

More specific information about individual routines listed in the left column in the following table is not available in this documentation. However, you can easily look up the information on a corresponding `SBCS` -, `_MBCS` -, or `_UNICODE` -related routine. Use the **Search** command on the **Help** menu to look up any generic-text routine listed below.

For related information, see Generic-Text Mappings in TCHAR.H.

Generic-Text Routine Mappings

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_cgetts`	`_cgets`	`_cgets`	`_cgetws`
`_cgetts_s`	`_cgets_s`	`_cgets_s`	`_cgetws_s`
`_cputts`	`_cputs`	`_cputs`	`_cputws`
`_fgettc`	`fgetc`	`fgetc`	`fgetwc`
`_fgettchar`	`_fgetchar`	`_fgetchar`	`_fgetwchar`
`_fgetts`	`fgets`	`fgets`	`fgetws`
`_fputtc`	`fputc`	`fputc`	`fputwc`
`_fputtchar`	`_fputchar`	`_fputchar`	`_fputwchar`
`_fputts`	`fputs`	`fputs`	`fputws`
`_ftprintf`	`fprintf`	`fprintf`	`fwprintf`
`_ftprintf_s`	`fprintf_s`	`fprintf_s`	`fwprintf_s`
`_ftscanf`	`fscanf`	`fscanf`	`fwscanf`
`_ftscanf_s`	`fscanf_s`	`fscanf_s`	`fwscanf_s`
`_gettc`	`getc`	`getc`	`getwc`
`_gettch`	`_getch`	`_getch`	`_getwch`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_gettchar`	`getchar`	`getchar`	`getwchar`
`_gettche`	`_getche`	`_getche`	`_getwche`
`_getts`	`gets`	`gets`	`getws`
`_getts_s`	`gets_s`	`gets_s`	`getws_s`
`_istalnum`	`isalnum`	`_ismbcalnum`	`iswalnum`
`_istalpha`	`isalpha`	`_ismbcalpha`	`iswalpha`
`_istascii`	`isascii`	`isascii`	`iswascii`
`_istcntrl`	`iscntrl`	`iscntrl`	`iswcntrl`
`_istdigit`	`isdigit`	`_ismbcdigit`	`iswdigit`
`_istgraph`	`isgraph`	`_ismbcgraph`	`iswgraph`
`_istlead`	Always returns false	`_ismblead`	Always returns false
`_istleadbyte`	Always returns false	`isleadbyte`	Always returns false
`_istlegal`	Always returns true	`_ismbclegal`	Always returns true
`_istlower`	`islower`	`_ismbclower`	`iswlower`
`_istprint`	`isprint`	`_ismbcprint`	`iswprint`
`_istpunct`	`ispunct`	`_ismbcpunct`	`iswpunct`
`_istspace`	`isspace`	`_ismbcspace`	`iswspace`
`_istupper`	`isupper`	`_ismbcupper`	`iswupper`
`_istxdigit`	`isxdigit`	`isxdigit`	`iswxdigit`
`_itot`	`_itoa`	`_itoa`	`_itow`
`_itot_s`	`_itoa_s`	`_itoa_s`	`_itow_s`
`_ltot`	`_ltoa`	`_ltoa`	`_ltow`
`_ltot_s`	`_ltoa_s`	`_ltoa_s`	`_ltow_s`
`_puttc`	`putc`	`putc`	`putwc`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_puttch`	`_putch`	`_putch`	`_putwch`
`_puttchar`	`putchar`	`putchar`	`putwchar`
`_putts`	`puts`	`puts`	`_putws`
`_sctprintf`	`_scprintf`	`_scprintf`	`_scwprintf`
`_sntprintf`	`_snprintf`	`_snprintf`	`_snwprintf`
`_sntprintf_s`	`_snprintf_s`	`_snprintf_s`	`_snwprintf_s`
`_sntscanf`	`_snscanf`	`_snscanf`	`_snwscanf`
`_sntscanf_s`	`_snscanf_s`	`_snscanf_s`	`_snwscanf_s`
`_stprintf`	`sprintf`	`sprintf`	`swprintf`
`_stprintf_s`	`sprintf_s`	`sprintf_s`	`swprintf_s`
`_stscanf`	`sscanf`	`sscanf`	`swscanf`
`_stscanf_s`	`sscanf_s`	`sscanf_s`	`swscanf_s`
`_taccess`	`_access`	`_access`	`_waccess`
`_taccess_s`	`_access_s`	`_access_s`	`_waccess_s`
`_tasctime`	`asctime`	`asctime`	`_wasctime`
`_tasctime_s`	`asctime_s`	`asctime_s`	`_wasctime_s`
`_tccmp`	Maps to macro or inline function	`_mbsncmp`	Maps to macro or inline function
`_tccpy`	Maps to macro or inline function	`_mbccpy`	Maps to macro or inline function
`_tccpy_s`	`strcpy_s`	`_mbccpy_s`	`wcscpy_s`
`_tchdir`	`_chdir`	`_chdir`	`_wchdir`
`_tclen`	Maps to macro or inline function	`_mbclen`	Maps to macro or inline function
`_tchmod`	`_chmod`	`_chmod`	`_wchmod`
`_tcprintf`	`_cprintf`	`_cprintf`	`_cwprintf`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
_tcprintf_s	_cprintf_s	_cprintf_s	_cwprintf_s
_tcreat	_creat	_creat	_wcreat
_tcscanf	_cscanf	_cscanf	_cwscanf
_tcscanf_s	_cscanf_s	_cscanf_s	_cwscanf_s
_tcscat	strcat	_mbscat	wcscat
_tcscat_s	strcat_s	_mbscat_s	wcscat_s
_tcschr	strchr	_mbschr	wcschr
_tcsclen	strlen	_mbslen	wcslen
_tcsclen_s	strlen_s	_mbslen_s	wcslen_s
_tcscmp	strcmp	_mbscmp	wcscmp
_tcscoll	strcoll	_mbscoll	wcscoll
_tcscpy	strcpy	_mbscpy	wcscpy
_tcscpy_s	strcpy_s	_mbscpy_s	wcscpy_s
_tcscspn	strcspn	_mbscspn	wcscspn
_tcsdec	_strdec	_mbsdec	_wcsdec
_tcsdup	_strdup	_mbsdup	_wcsdup
_tcserror	strerror	strerror	_wcserror
_tcserror_s	strerror_s	strerror_s	_wcserror_s
_tcsftime	strftime	strftime	wcsftime
_tcsicmp	_stricmp	_mbsicmp	_wcsicmp
_tcsicoll	_stricoll	_mbsicoll	_wcsicoll
_tcsinc	_strinc	_mbsinc	_wcsinc
_tcslen	strlen	strlen	wcslen
_tcslwr	_strlwr	_mbslwr	_wcslwr

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
_tcslwr_s	_strlwr_s	_mbslwr_s	_wcslwr_s
_tcsnbcnt	_strncnt	_mbsnbcnt	_wcsncnt
_tcsncat	strncat	_mbsnbcat	wcsncat
_tcsncat_s	strncat_s	_mbsnbcat_s	wcsncat_s
_tcsnccat	strncat	_mbsncat	wcsncat
_tcsnccmp	strncmp	_mbsncmp	wcsncmp
_tcsnccmp_s	strncmp_s	_mbsncmp_s	wcsncmp_s
_tcsnccoll	_strncoll	_mbsncoll	_wcsncoll
_tcsncmp	strncmp	_mbsnbcmp	wcsncmp
_tcsnccnt	_strncnt	_mbsnccnt	_wcsncnt
_tcsnccpy	strncpy	_mbsncpy	wcsncpy
_tcsnccpy_s	strncpy_s	_mbsncpy_s	wcsncpy_s
_tcsncicmp	_strnicmp	_mbsnicmp	_wcsnicmp
_tcsncicoll	_strnicoll	_mbsnicoll	_wcsnicoll
_tcsncpy	strncpy	_mbsnbcpy	wcsncpy
_tcsncpy_s	strncpy_s	_mbsnbcpy_s	wcsncpy_s
_tcsncset	_strnset	_mbsnset	_wcsnset
_tcsnextc	_strnextc	_mbsnextc	_wcsnextc
_tcsnicmp	_strnicmp	_mbsnbicmp	_wcsnicmp
_tcsnicoll	_strnicoll	_mbsnbicoll	_wcsnicoll
_tcsninc	_strninc	_mbsninc	_wcsninc
_tcsnccnt	_strncnt	_mbsnccnt	_wcsncnt
_tcsnset	_strnset	_mbsnbset	_wcsnset
_tcspbrk	strpbrk	_mbspbrk	wcspbrk

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
_tcsspnp	_strspnp	_mbsspnp	_wcsspnp
_tcsrchr	strrchr	_mbsrchr	wcsrchr
_tcsrev	_strrev	_mbsrev	_wcsrev
_tcsset	_strset	_mbsset	_wcsset
_tcsspn	strspn	_mbsspn	wcsspn
_tcsstr	strstr	_mbsstr	wcsstr
_tcstod	strtod	strtod	wcstod
_tcstoi64	_strtoi64	_strtoi64	_wcstoi64
_tcstok	strtok	_mbstok	wcstok
_tcstok_s	strtok_s	_mbstok_s	wcstok_s
_tcstol	strtol	strtol	wcstol
_tcstoui64	_strtoui64	_strtoui64	_wcstoui64
_tcstoul	strtoul	strtoul	wcstoul
_tcsupr	_strupr	_mbsupr	_wcsupr
_tcsupr_s	_strupr_s	_mbsupr_s	_wcsupr_s
_tcsxfrm	strxfrm	strxfrm	wcsxfrm
_tctime	ctime	ctime	_wctime
_tctime_s	ctime_s	ctime_s	_wctime_s
_tctime32	_ctime32	_ctime32	_wctime32
_tctime32_s	_ctime32_s	_ctime32_s	_wctime32_s
_tctime64	_ctime64	_ctime64	_wctime64
_tctime64_s	_ctime64_s	_ctime64_s	_wctime64_s
_texecl	_execl	_execl	_wexecl
_texecle	_execle	_execle	_wexecle

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_texeclp`	`_execlp`	`_execlp`	`_wexeclp`
`_texeclpe`	`_execlpe`	`_execlpe`	`_wexeclpe`
`_texecv`	`_execv`	`_execv`	`_wexecv`
`_texecve`	`_execve`	`_execve`	`_wexecve`
`_texecvp`	`_execvp`	`_execvp`	`_wexecvp`
`_texecvpe`	`_execvpe`	`_execvpe`	`_wexecvpe`
`_tfdopen`	`_fdopen`	`_fdopen`	`_wfdopen`
`_tfindfirst`	`_findfirst`	`_findfirst`	`_wfindfirst`
`_tfindnext`	`_findnext`	`_findnext`	`_wfindnext`
`_tfindnext32`	`_findnext32`	`_findnext32`	`_wfindnext32`
`_tfindnext64`	`_findnext64`	`_findnext64`	`_wfindnext64`
`_tfindnexti64`	`_findnexti64`	`_findnexti64`	`_wfindnexti64`
`_tfindnexti6432`	`_findnexti6432`	`_findnexti6432`	`_wfindnexti6432`
`_tfindnext32i64`	`_findnext32i64`	`_findnext32i64`	`_wfindnext32i64`
`_tfopen`	`fopen`	`fopen`	`_wfopen`
`_tfopen_s`	`fopen_s`	`fopen_s`	`_wfopen_s`
`_tfreopen`	`freopen`	`freopen`	`_wfreopen`
`_tfreopen_s`	`freopen_s`	`freopen_s`	`_wfreopen_s`
`_tfsopen`	`_fsopen`	`_fsopen`	`_wfsopen`
`_tfullpath`	`_fullpath`	`_fullpath`	`_wfullpath`
`_tgetcwd`	`_getcwd`	`_getcwd`	`_wgetcwd`
`_tgetdcwd`	`_getdcwd`	`_getdcwd`	`_wgetdcwd`
`_tgetenv`	`getenv`	`getenv`	`_wgetenv`
`_tgetenv_s`	`getenv_s`	`getenv_s`	`_wgetenv_s`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_tmain`	`main`	`main`	`wmain`
`_tmakepath`	`_makepath`	`_makepath`	`_wmakepath`
`_tmakepath_s`	`_makepath_s`	`_makepath_s`	`_wmakepath_s`
`_tmkdir`	`_mkdir`	`_mkdir`	`_wmkdir`
`_tmktemp`	`_mktemp`	`_mktemp`	`_wmktemp`
`_tmktemp_s`	`_mktemp_s`	`_mktemp_s`	`_wmktemp_s`
`_topen`	`_open`	`_open`	`_wopen`
`_topen_s`	`_open_s`	`_open_s`	`_wopen_s`
`_totlower`	`tolower`	`_mbctolower`	`towlower`
`_totupper`	`toupper`	`_mbctoupper`	`towupper`
`_tperror`	`perror`	`perror`	`_wperror`
`_tpopen`	`_popen`	`_popen`	`_wpopen`
`_tprintf`	`printf`	`printf`	`wprintf`
`_tprintf_s`	`printf_s`	`printf_s`	`wprintf_s`
`_tputenv`	`_putenv`	`_putenv`	`_wputenv`
`_tputenv_s`	`_putenv_s`	`_putenv_s`	`_wputenv_s`
`_tremove`	`remove`	`remove`	`_wremove`
`_trename`	`rename`	`rename`	`_wrename`
`_trmdir`	`_rmdir`	`_rmdir`	`_wrmdir`
`_tsearchenv`	`_searchenv`	`_searchenv`	`_wsearchenv`
`_tsearchenv_s`	`_searchenv_s`	`_searchenv_s`	`_wsearchenv_s`
`_tscanf`	`scanf`	`scanf`	`wscanf`
`_tscanf_s`	`scanf_s`	`scanf_s`	`wscanf_s`
`_tsetlocale`	`setlocale`	`setlocale`	`_wsetlocale`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
_tsopen	_sopen	_sopen	_wsopen
_tsopen_s	_sopen_s	_sopen_s	_wsopen_s
_tspawnl	_spawnl	_spawnl	_wspawnl
_tspawnle	_spawnle	_spawnle	_wspawnle
_tspawnlp	_spawnlp	_spawnlp	_wspawnlp
_tspawnlpe	_spawnlpe	_spawnlpe	_wspawnlpe
_tspawnv	_spawnv	_spawnv	_wspawnv
_tspawnve	_spawnve	_spawnve	_wspawnve
_tspawnvp	_spawnvp	_spawnvp	_wspawnvp
_tspawnvpe	_spawnvpe	_spawnvpe	_wspawnvpe
_tsplitpath	_splitpath	_splitpath	_wsplitpath
_tstat	_stat	_stat	_wstat
_tstat32	_stat32	_stat32	_wstat32
_tstati32	_stati32	_stati32	_wstati32
_tstat64	_stat64	_stat64	_wstat64
_tstati64	_stati64	_stati64	_wstati64
_tstof	atof	atof	_wtof
_tstoi	atoi	atoi	_wtoi
_tstoi64	_atoi64	_atoi64	_wtoi64
_tstol	atol	atol	_wtol
_tstrdate	_strdate	_strdate	_wstrdate
_tstrdate_s	_strdate_s	_strdate_s	_wstrdate_s
_tstrtime	_strtime	_strtime	_wstrtime
_tstrtime_s	_strtime_s	_strtime_s	_wstrtime_s

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_tsystem`	`system`	`system`	`_wsystem`
`_ttempnam`	`_tempnam`	`_tempnam`	`_wtempnam`
`_ttmpnam`	`tmpnam`	`tmpnam`	`_wtmpnam`
`_ttmpnam_s`	`tmpnam_s`	`tmpnam_s`	`_wtmpnam_s`
`_ttoi`	`atoi`	`atoi`	`_wtoi`
`_ttoi64`	`_atoi64`	`_atoi64`	`_wtoi64`
`_ttol`	`atol`	`atol`	`_wtol`
`_tunlink`	`_unlink`	`_unlink`	`_wunlink`
`_tutime`	`_utime`	`_utime`	`_wutime`
`_tutime32`	`_utime32`	`_utime32`	`_wutime32`
`_tutime64`	`_utime64`	`_utime64`	`_wutime64`
`_tWinMain`	`WinMain`	`WinMain`	`wWinMain`
`_ui64tot`	`_ui64toa`	`_ui64toa`	`_ui64tow`
`_ui64tot_s`	`_ui64toa_s`	`_ui64toa_s`	`_ui64tow_s`
`_ultot`	`_ultoa`	`_ultoa`	`_ultow`
`_ultot_s`	`_ultoa_s`	`_ultoa_s`	`_ultow_s`
`_ungettc`	`ungetc`	`ungetc`	`ungetwc`
`_ungettch`	`_ungetch`	`_ungetch`	`_ungetwch`
`_vftprintf`	`vfprintf`	`vfprintf`	`vfwprintf`
`_vftprintf_s`	`vfprintf_s`	`vfprintf_s`	`vfwprintf_S`
`_vsctprintf`	`_vscprintf`	`_vscprintf`	`_vscwprintf`
`_vsctprintf_s`	`_vscprintf_s`	`_vscprintf_s`	`_vscwprintf_S`
`_vsntprintf`	`_vsnprintf`	`_vsnprintf`	`_vsnwprintf`
`_vsntprintf_s`	`_vsnprintf_s`	`_vsnprintf_s`	`_vsnwprintf_s`

GENERIC-TEXT ROUTINE NAME	SBCS (_UNICODE & MBCS NOT DEFINED)	_MBCS DEFINED	_UNICODE DEFINED
`_vstprintf`	`vsprintf`	`vsprintf`	`vswprintf`
`_vstprintf_s`	`vsprintf_s`	`vsprintf_s`	`vswprintf_s`
`_vtprintf`	`vprintf`	`vprintf`	`vwprintf`
`_vtprintf_s`	`vprintf_s`	`vprintf_s`	`vwprintf_s`

See also

Generic-Text Mappings
Data Type Mappings
Constant and Global Variable Mappings
A Sample Generic-Text Program
Using Generic-Text Mappings

UCRT Locale names, Languages, and Country/Region strings

12/11/2018 • 3 minutes to read • Edit Online

The *locale* argument to the setlocale, _wsetlocale, _create_locale, and _wcreate_locale functions can be set by using the locale names, languages, country/region codes, and code pages that are supported by the Windows NLS API. The *locale* argument takes the following form:

```
locale :: "locale-name"
   | "language[_country-region[.code-page]]"
   | ".code-page"
   | "C"
   | ""
   | NULL
```

The *locale-name* form is a short, IETF-standardized string; for example, `en-US` for English (United States) or `bs-Cyrl-BA` for Bosnian (Cyrillic, Bosnia and Herzegovina). These forms are preferred. For a list of supported locale names by Windows operating system version, see the **Language tag** column of the table in Appendix A: Product Behavior in [MS-LCID]: Windows Language Code Identifier (LCID) Reference. This resource lists the supported language, script, and region parts of the locale names. For information about the supported locale names that have non-default sort orders, see the **Locale name** column in Sort Order Identifiers. Under Windows 10 or later, locale names that correspond to valid BCP-47 language tags are allowed. For example, `jp-US` is a valid BCP-47 tag, but it is effectively only `US` for locale functionality.

The *language*[*_country-region*[**.***code-page*]] form is stored in the locale setting for a category when a language string, or language string and country or region string, is used to create the locale. The set of supported language strings is described in Language Strings, and the list of supported country and region strings is listed in Country/Region Strings. If the specified language is not associated with the specified country or region, the default language for the specified country or region is stored in the locale setting. We do not recommend this form for locale strings embedded in code or serialized to storage, because these strings are more likely to be changed by an operating system update than the locale name form.

The *code-page* is the ANSI/OEM code page that's associated with the locale. The code page is determined for you when you specify a locale by language or by language and country/region alone. The special value `.ACP` specifies the ANSI code page for the country/region. The special value `.OCP` specifies the OEM code page for the country/region. For example, if you specify `"Greek_Greece.ACP"` as the locale, the locale is stored as `Greek_Greece.1253` (the ANSI code page for Greek), and if you specify `"Greek_Greece.OCP"` as the locale, it is stored as `Greek_Greece.737` (the OEM code page for Greek). For more information about code pages, see Code Pages. For a list of supported code pages on Windows, see Code Page Identifiers.

If you use only the code page to specify the locale, the user's default language and country/region as reported by GetUserDefaultLocaleName are used. For example, if you specify `".1254"` (ANSI Turkish) as the locale for a user that's configured for English (United States), the locale that's stored is `English_United States.1254`. We do not recommend this form, because it could lead to inconsistent behavior.

A *locale* argument value of `C` specifies the minimal ANSI conforming environment for C translation. The `C` locale assumes that every **char** data type is 1 byte and its value is always less than 256. If *locale* points to an empty string, the locale is the implementation-defined native environment.

You can specify all of the locale categories at the same time for the `setlocale` and `_wsetlocale` functions by using the `LC_ALL` category. The categories can all be set to the same locale, or you can set each category individually by using a locale argument that has this form:

> *LC-ALL-specifier* :: *locale*
> | [**LC_COLLATE=***locale*][**;LC_CTYPE=***locale*][**;LC_MONETARY=***locale*][**;LC_NUMERIC=***locale*]
> [**;LC_TIME=***locale*]

You can specify multiple category types, separated by semicolons. Category types that are not specified use the current locale setting. For example, this code snippet sets the current locale for all categories to `de-DE`, and then sets the categories `LC_MONETARY` to `en-GB` and `LC_TIME` to `es-ES`:

```
_wsetlocale(LC_ALL, L"de-DE");
_wsetlocale(LC_ALL, L"LC_MONETARY=en-GB;LC_TIME=es-ES");
```

See also

C Run-Time Library Reference
_get_current_locale
setlocale, _wsetlocale
_create_locale, _wcreate_locale
Language Strings
Country/Region Strings

Language Strings

10/31/2018 • 2 minutes to read • Edit Online

The setlocale and _create_locale functions can use the Windows NLS API supported languages on operating systems that do not use the Unicode code page. For a list of supported languages by operating system version, see Appendix A: Product Behavior in [MS-LCID]: Windows Language Code Identifier (LCID) Reference. The language string can be any of the values in the **Language** and **Language tag** columns of the list of supported languages. For an example of code that enumerates available locale names and related values, see NLS: Name-based APIs Sample.

Additional supported language strings

The Microsoft C run-time library implementation also supports these language strings:

LANGUAGE STRING	EQUIVALENT LOCALE NAME
american	en-US
american english	en-US
american-english	en-US
australian	en-AU
belgian	nl-BE
canadian	en-CA
chh	zh-HK
chi	zh-SG
chinese	zh
chinese-hongkong	zh-HK
chinese-simplified	zh-CN
chinese-singapore	zh-SG
chinese-traditional	zh-TW
dutch-belgian	nl-BE
english-american	en-US
english-aus	en-AU
english-belize	en-BZ

LANGUAGE STRING	EQUIVALENT LOCALE NAME
english-can	en-CA
english-caribbean	en-029
english-ire	en-IE
english-jamaica	en-JM
english-nz	en-NZ
english-south africa	en-ZA
english-trinidad y tobago	en-TT
english-uk	en-GB
english-us	en-US
english-usa	en-US
french-belgian	fr-BE
french-canadian	fr-CA
french-luxembourg	fr-LU
french-swiss	fr-CH
german-austrian	de-AT
german-lichtenstein	de-LI
german-luxembourg	de-LU
german-swiss	de-CH
irish-english	en-IE
italian-swiss	it-CH
norwegian	no
norwegian-bokmal	nb-NO
norwegian-nynorsk	nn-NO
portuguese-brazilian	pt-BR
spanish-argentina	es-AR

LANGUAGE STRING	EQUIVALENT LOCALE NAME
spanish-bolivia	es-BO
spanish-chile	es-CL
spanish-colombia	es-CO
spanish-costa rica	es-CR
spanish-dominican republic	es-DO
spanish-ecuador	es-EC
spanish-el salvador	es-SV
spanish-guatemala	es-GT
spanish-honduras	es-HN
spanish-mexican	es-MX
spanish-modern	es-ES
spanish-nicaragua	es-NI
spanish-panama	es-PA
spanish-paraguay	es-PY
spanish-peru	es-PE
spanish-puerto rico	es-PR
spanish-uruguay	es-UY
spanish-venezuela	es-VE
swedish-finland	sv-FI
swiss	de-CH
uk	en-GB
us	en-US
usa	en-US

See also

Locale Names, Languages, and Country/Region Strings
Country/Region Strings

setlocale, _wsetlocale
_create_locale, _wcreate_locale

Country/Region Strings

10/31/2018 • 2 minutes to read • Edit Online

Country and region strings can be combined with a language string to create a locale specification for the `setlocale`, `_wsetlocale`, `_create_locale`, and `_wcreate_locale` functions. For lists of country and region names that are supported by various Windows operating system versions, see the **Language**, **Location**, and **Language tag** columns of the table in Appendix A: Product Behavior in [MS-LCID]: Windows Language Code Identifier (LCID) Reference. For an example of code that enumerates available locale names and related values, see NLS: Name-based APIs Sample.

Additional supported country and region strings

The Microsoft C run-time library implementation also supports the following additional country/region strings and abbreviations:

COUNTRY/REGION STRING	ABBREVIATION	EQUIVALENT LOCALE NAME
america	USA	en-US
britain	GBR	en-GB
china	CHN	zh-CN
czech	CZE	cs-CZ
england	GBR	en-GB
great britain	GBR	en-GB
holland	NLD	nl-NL
hong-kong	HKG	zh-HK
new-zealand	NZL	en-NZ
nz	NZL	en-NZ
pr china	CHN	zh-CN
pr-china	CHN	zh-CN
puerto-rico	PRI	es-PR
slovak	SVK	sk-SK
south africa	ZAF	af-ZA
south korea	KOR	ko-KR

COUNTRY/REGION STRING	ABBREVIATION	EQUIVALENT LOCALE NAME
south-africa	ZAF	af-ZA
south-korea	KOR	ko-KR
trinidad & tobago	TTO	en-TT
uk	GBR	en-GB
united-kingdom	GBR	en-GB
united-states	USA	en-US
us	USA	en-US

See also

Locale Names, Languages, and Country/Region Strings
Language Strings
setlocale, _wsetlocale
_create_locale, _wcreate_locale

Function Family Overviews

10/31/2018 • 2 minutes to read • Edit Online

Insert introduction here.

Section Heading

Insert section body here.

Subsection Heading

Insert subsection body here.

_exec, _wexec Functions

3/11/2019 • 7 minutes to read • Edit Online

Each function in this family loads and executes a new process:

_execl, _wexecl	_execv, _wexecv
_execle, _wexecle	_execve, _wexecve
_execlp, _wexeclp	_execvp, _wexecvp
_execlpe, _wexeclpe	_execvpe, _wexecvpe

The letter at the end of the function name determines the variation.

_EXEC FUNCTION SUFFIX	DESCRIPTION
`e`	`envp`, array of pointers to environment settings, is passed to the new process.
`l`	Command-line arguments are passed individually to `_exec` function. Typically used when the number of parameters to the new process is known in advance.
`p`	`PATH` environment variable is used to find the file to execute.
`v`	`argv`, array of pointers to command-line arguments, is passed to `_exec`. Typically used when the number of parameters to the new process is variable.

Remarks

Each `_exec` function loads and executes a new process. All `_exec` functions use the same operating-system function (CreateProcess). The `_exec` functions automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. The `_wexec` functions are wide-character versions of the `_exec` functions. The `_wexec` functions behave identically to their `_exec` family counterparts except that they do not handle multibyte-character strings.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_texecl`	`_execl`	`_execl`	`_wexecl`
`_texecle`	`_execle`	`_execle`	`_wexecle`
`_texeclp`	`_execlp`	`_execlp`	`_wexeclp`

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_texeclpe`	`_execlpe`	`_execlpe`	`_wexeclpe`
`_texecv`	`_execv`	`_execv`	`_wexecv`
`_texecve`	`_execve`	`_execve`	`_wexecve`
`_texecvp`	`_execvp`	`_execvp`	`_wexecvp`
`_texecvpe`	`_execvpe`	`_execvpe`	`_wexecvpe`

The `cmdname` parameter specifies the file to be executed as the new process. It can specify a full path (from the root), a partial path (from the current working directory), or a file name. If `cmdname` does not have a file name extension or does not end with a period (.), the `_exec` function searches for the named file. If the search is unsuccessful, it tries the same base name with the .com file name extension and then with the .exe, .bat, and .cmd file name extensions. If `cmdname` has a file name extension, only that extension is used in the search. If `cmdname` ends with a period, the `_exec` function searches for `cmdname` with no file name extension. `_execlp`, `_execlpe`, `_execvp`, and `_execvpe` search for `cmdname` (using the same procedures) in the directories specified by the `PATH` environment variable. If `cmdname` contains a drive specifier or any slashes (that is, if it is a relative path), the `_exec` call searches only for the specified file; the path is not searched.

Parameters are passed to the new process by giving one or more pointers to character strings as parameters in the `_exec` call. These character strings form the parameter list for the new process. The combined length of the inherited environment settings and the strings forming the parameter list for the new process must not exceed 32 kilobytes. The terminating null character ('\0') for each string is not included in the count, but space characters (inserted automatically to separate the parameters) are counted.

NOTE

Spaces embedded in strings may cause unexpected behavior; for example, passing `_exec` the string `"hi there"` will result in the new process getting two arguments, `"hi"` and `"there"`. If the intent was to have the new process open a file named "hi there", the process would fail. You can avoid this by quoting the string: `"\"hi there\""`.

IMPORTANT

Do not pass user input to `_exec` without explicitly checking its content. `_exec` will result in a call to CreateProcess so keep in mind that unqualified path names could lead to potential security vulnerabilities.

The `_exec` functions validate their parameters. If expected parameters are null pointers, empty strings, or omitted, the `_exec` functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set `errno` to `EINVAL` and return -1. No new process is executed.

The argument pointers can be passed as separate parameters (in `_execl`, `_execle`, `_execlp`, and `_execlpe`) or as an array of pointers (in `_execv`, `_execve`, `_execvp`, and `_execvpe`). At least one parameter, `arg0`, must be passed to the new process; this parameter is `argv`[0] of the new process. Usually, this parameter is a copy of `cmdname`. (A different value does not produce an error.)

The `_execl`, `_execle`, `_execlp`, and `_execlpe` calls are typically used when the number of parameters is known in advance. The parameter `arg0` is usually a pointer to `cmdname`. The parameters `arg1` through `argn`

point to the character strings forming the new parameter list. A null pointer must follow `argn` to mark the end of the parameter list.

The `_execv`, `_execve`, `_execvp`, and `_execvpe` calls are useful when the number of parameters to the new process is variable. Pointers to the parameters are passed as an array, `argv`. The parameter `argv`[0] is usually a pointer to `cmdname`. The parameters `argv`[1] through `argv`[`n`] point to the character strings forming the new parameter list. The parameter `argv`[`n`+1] must be a **NULL** pointer to mark the end of the parameter list.

Files that are open when an `_exec` call is made remain open in the new process. In `_execl`, `_execlp`, `_execv`, and `_execvp` calls, the new process inherits the environment of the calling process. `_execle`, `_execlpe`, `_execve`, and `_execvpe` calls alter the environment for the new process by passing a list of environment settings through the `envp` parameter. `envp` is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form `NAME`=`value` where `NAME` is the name of an environment variable and `value` is the string value to which that variable is set. (Note that `value` is not enclosed in double quotation marks.) The final element of the `envp` array should be **NULL**. When `envp` itself is **NULL**, the new process inherits the environment settings of the calling process.

A program executed with one of the `_exec` functions is always loaded into memory as if the maximum allocation field in the program's .exe file header were set to the default value of 0xFFFFH.

The `_exec` calls do not preserve the translation modes of open files. If the new process must use files inherited from the calling process, use the _setmode routine to set the translation mode of these files to the desired mode. You must explicitly flush (using `fflush` or `_flushall`) or close any stream before the `_exec` function call. Signal settings are not preserved in new processes that are created by calls to `_exec` routines. The signal settings are reset to the default in the new process.

Example

```
// crt_args.c
// Illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
// This program will be executed by crt_exec which follows.

#include <stdio.h>

int main( int argc,  // Number of strings in array argv
char *argv[],       // Array of command-line argument strings
char **envp )       // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf( "  %s\n", *(envp++) );

    return;
}
```

Run the following program to execute Crt_args.exe:

```
// crt_exec.c
// Illustrates the different versions of exec, including
//      _execl          _execle          _execlp          _execlpe
//      _execv          _execve          _execvp          _execvpe
//
// Although CRT_EXEC.C can exec any program, you can verify how
// different versions handle arguments and environment by
// compiling and specifying the sample program CRT_ARGS.C. See
// "_spawn, _wspawn Functions" for examples of the similar spawn
// functions.

#include <stdio.h>
#include <conio.h>
#include <process.h>

char *my_env[] =     // Environment for exec?e
{
   "THIS=environment will be",
   "PASSED=to new process by",
   "the EXEC=functions",
   NULL
};

int main( int ac, char* av[] )
{
   char *args[4];
   int ch;
   if( ac != 3 ){
      fprintf( stderr, "Usage: %s <program> <number (1-8)>\n", av[0] );
      return;
   }

   // Arguments for _execv?
   args[0] = av[1];
   args[1] = "exec??";
   args[2] = "two";
   args[3] = NULL;

   switch( atoi( av[2] ) )
   {
   case 1:
      _execl( av[1], av[1], "_execl", "two", NULL );
      break;
   case 2:
      _execle( av[1], av[1], "_execle", "two", NULL, my_env );
      break;
   case 3:
      _execlp( av[1], av[1], "_execlp", "two", NULL );
      break;
   case 4:
      _execlpe( av[1], av[1], "_execlpe", "two", NULL, my_env );
      break;
   case 5:
      _execv( av[1], args );
      break;
   case 6:
      _execve( av[1], args, my_env );
      break;
   case 7:
      _execvp( av[1], args );
      break;
   case 8:
      _execvpe( av[1], args, my_env );
      break;
   default:
      break;
   }
```

```
    // This point is reached only if exec fails.
    printf( "\nProcess was not execed." );
    exit( 0 );
}
```

Requirements

Header: process.h

See also

Process and Environment Control
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

Filename Search Functions

3/11/2019 • 4 minutes to read • Edit Online

These functions search for and close searches for specified file names:

- _findnext, _wfindnext
- _findfirst, _wfindfirst
- _findclose

Remarks

The `_findfirst` function provides information about the first instance of a file name that matches the file specified in the `filespec` argument. You can use in `filespec` any combination of wildcard characters that is supported by the host operating system.

The functions return file information in a `_finddata_t` structure, which is defined in IO.h. Various functions in the family use many variations on the `_finddata_t` structure. The basic `_finddata_t` structure includes the following elements:

`unsigned attrib`
File attribute.

`time_t time_create`
Time of file creation (-1L for FAT file systems). This time is stored in UTC format. To convert to the local time, use localtime_s.

`time_t time_access`
Time of the last file access (-1L for FAT file systems). This time is stored in UTC format. To convert to the local time, use `localtime_s`.

`time_t time_write`
Time of the last write to file. This time is stored in UTC format. To convert to the local time, use `localtime_s`.

`_fsize_t size`
Length of the file in bytes.

`char name`[`_MAX_PATH`] Null-terminated name of matched file or directory, without the path.

In file systems that do not support the creation and last access times of a file, such as the FAT system, the `time_create` and `time_access` fields are always -1L.

`_MAX_PATH` is defined in Stdlib.h as 260 bytes.

You cannot specify target attributes (such as `_A_RDONLY`) to limit the find operation. These attributes are returned in the `attrib` field of the `_finddata_t` structure and can have the following values (defined in IO.h). Users should not rely on these being the only values possible for the `attrib` field.

`_A_ARCH`
Archive. Set whenever the file is changed and cleared by the **BACKUP** command. Value: 0x20.

`_A_HIDDEN`
Hidden file. Not generally seen with the DIR command, unless you use the **/AH** option. Returns information

about normal files and files that have this attribute. Value: 0x02.

`_A_NORMAL`
Normal. File has no other attributes set and can be read or written to without restriction. Value: 0x00.

`_A_RDONLY`
Read-only. File cannot be opened for writing and a file that has the same name cannot be created. Value: 0x01.

`_A_SUBDIR`
Subdirectory. Value: 0x10.

`_A_SYSTEM`
System file. Not ordinarily seen with the **DIR** command, unless the **/A** or **/A:S** option is used. Value: 0x04.

`_findnext` finds the next name, if any, that matches the `filespec` argument specified in an earlier call to `_findfirst`. The `fileinfo` argument should point to a structure initialized by the previous call to `_findfirst`. If a match is found, the `fileinfo` structure contents are changed as described earlier. Otherwise, it is left unchanged. `_findclose` closes the specified search handle and releases all associated resources for both `_findfirst` and `_findnext`. The handle returned by either `_findfirst` or `_findnext` must first be passed to `_findclose`, before modification operations, such as deleting, can be performed on the directories that form the paths passed to them.

You can nest the `_find` functions. For example, if a call to `_findfirst` or `_findnext` finds the file that is a subdirectory, a new search can be initiated with another call to `_findfirst` or `_findnext`.

`_wfindfirst` and `_wfindnext` are wide-character versions of `_findfirst` and `_findnext`. The structure argument of the wide-character versions has the `_wfinddata_t` data type, which is defined in IO.h and in Wchar.h. The fields of this data type are the same as those of the `_finddata_t` data type, except that in `_wfinddata_t` the name field is of type `wchar_t` instead of type `char`. Otherwise `_wfindfirst` and `_wfindnext` behave identically to `_findfirst` and `_findnext`.

`_findfirst` and `_findnext` use the 64-bit time type. If you must use the old 32-bit time type, you can define `_USE_32BIT_TIME_T`. The versions of these functions that have the `32` suffix in their names use the 32-bit time type, and those with the `64` suffix use the 64-bit time type.

Functions `_findfirst32i64`, `_findnext32i64`, `_wfindfirst32i64`, and `_wfindnext32i64` also behave identically to the 32-bit time type versions of these functions except they use and return 64-bit file lengths. Functions `_findfirst64i32`, `_findnext64i32`, `_wfindfirst64i32`, and `_wfindnext64i32` use the 64-bit time type but use 32-bit file lengths. These functions use appropriate variations of the `_finddata_t` type in which the fields have different types for the time and the file size.

`_finddata_t` is actually a macro that evaluates to `_finddata64i32_t` (or `_finddata32_t` if `_USE_32BIT_TIME_T` is defined). The following table summarizes the variations on `_finddata_t`:

STRUCTURE	TIME TYPE	FILE SIZE TYPE
`_finddata_t`, `_wfinddata_t`	`__time64_t`	`_fsize_t`
`_finddata32_t`, `_wfinddata32_t`	`__time32_t`	`_fsize_t`
`__finddata64_t`, `__wfinddata64_t`	`__time64_t`	`__int64`
`_finddata32i64_t`, `_wfinddata32i64_t`	`__time32_t`	`__int64`

STRUCTURE	TIME TYPE	FILE SIZE TYPE
`_finddata64i32_t`, `_wfinddata64i32_t`	`__time64_t`	`_fsize_t`

`_fsize_t` is a `typedef` for `unsigned long` (32 bits).

Example

```
// crt_find.c
// This program uses the 32-bit _find functions to print
// a list of all files (and their attributes) with a .C extension
// in the current directory.

#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <time.h>

int main( void )
{
   struct _finddata_t c_file;
   intptr_t hFile;

   // Find first .c file in current directory
   if( (hFile = _findfirst( "*.c", &c_file )) == -1L )
      printf( "No *.c files in current directory!\n" );
   else
   {
      printf( "Listing of .c files\n\n" );
      printf( "RDO HID SYS ARC  FILE         DATE %25c SIZE\n", ' ' );
      printf( "--- --- --- ---  ----         ---- %25c ----\n", ' ' );
      do {
         char buffer[30];
         printf( ( c_file.attrib & _A_RDONLY ) ? " Y  " : " N  " );
         printf( ( c_file.attrib & _A_HIDDEN ) ? " Y  " : " N  " );
         printf( ( c_file.attrib & _A_SYSTEM ) ? " Y  " : " N  " );
         printf( ( c_file.attrib & _A_ARCH )   ? " Y  " : " N  " );
         ctime_s( buffer, _countof(buffer), &c_file.time_write );
         printf( " %-12s %.24s  %9ld\n",
            c_file.name, buffer, c_file.size );
      } while( _findnext( hFile, &c_file ) == 0 );
      _findclose( hFile );
   }
}
```

```
Listing of .c files

RDO HID SYS ARC  FILE         DATE                          SIZE
--- --- --- ---  ----         ----                          ----
N   N   N   Y   blah.c       Wed Feb 13 09:21:42 2002       1715
N   N   N   Y   test.c       Wed Feb 06 14:30:44 2002        312
```

See also

System Calls

Format specification syntax: printf and wprintf functions

3/11/2019 • 16 minutes to read • Edit Online

The various `printf` and `wprintf` functions take a format string and optional arguments and produce a formatted sequence of characters for output. The format string contains zero or more *directives*, which are either literal characters for output or encoded *conversion specifications* that describe how to format an argument in the output. This topic describes the syntax used to encode conversion specifications in the format string. For a listing of these functions, see Stream I/O.

A conversion specification consists of optional and required fields in this form:

%[*flags*][*width*][.*precision*][*size*]*type*

Each field of the conversion specification is a character or a number that signifies a particular format option or conversion specifier. The required *type* field specifies the kind of conversion to be applied to an argument. The optional *flags*, *width*, and *precision* fields control additional format aspects such as leading spaces or zeroes, justification, and displayed precision. The *size* field specifies the size of the argument consumed and converted.

A basic conversion specification contains only the percent sign and a *type* character. For example, `%s` specifies a string conversion. To print a percent-sign character, use `%%`. If a percent sign is followed by a character that has no meaning as a format field, the invalid parameter handler is invoked. For more information, see Parameter Validation.

IMPORTANT

For security and stability, ensure that conversion specification strings are not user-defined. For example, consider a program that prompts the user to enter a name and stores the input in a string variable that's named `user_name`. To print `user_name`, do not do this:

```
printf( user_name ); /* Danger! If user_name contains "%s", program will crash */
```

Instead, do this:

```
printf( "%s", user_name );
```

Type conversion specifier

The *type* conversion specifier character specifies whether to interpret the corresponding argument as a character, a string, a pointer, an integer, or a floating-point number. The *type* character is the only required conversion specification field, and it appears after any optional fields.

The arguments that follow the format string are interpreted according to the corresponding *type* character and the optional size prefix. Conversions for character types `char` and `wchar_t` are specified by using **c** or **C**, and single-byte and multi-byte or wide character strings are specified by using **s** or **S**, depending on which formatting function is being used. Character and string arguments that are specified by using **c** and **s** are interpreted as `char` and `char*` by `printf` family functions, or as `wchar_t` and `wchar_t*` by `wprintf` family functions. Character and string arguments that are specified by using **C** and **S** are interpreted as `wchar_t` and `wchar_t*` by `printf` family functions, or as `char` and `char*` by `wprintf` family functions. This behavior is Microsoft specific.

Integer types such as `short`, `int`, `long`, `long long`, and their `unsigned` variants, are specified by using **d**, **i**, **o**, **u**, **x**, and **X**. Floating-point types such as `float`, `double`, and `long double`, are specified by using **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G**. By default, unless they are modified by a *size* prefix, integer arguments are coerced to `int` type, and floating-point arguments are coerced to `double`. On 64-bit systems, an `int` is a 32-bit value; therefore, 64-bit integers will be truncated when they are formatted for output unless a *size* prefix of **ll** or **I64** is used. Pointer types that are specified by **p** use the default pointer size for the platform.

> **NOTE**
>
> **Microsoft Specific** The **Z** type character, and the behavior of the **c**, **C**, **s**, and **S** type characters when they are used with the `printf` and `wprintf` functions, are Microsoft extensions. The ISO C standard uses **c** and **s** consistently for narrow characters and strings, and **C** and **S** for wide characters and strings, in all formatting functions.

Type field characters

TYPE CHARACTER	ARGUMENT	OUTPUT FORMAT
c	Character	When used with `printf` functions, specifies a single-byte character; when used with `wprintf` functions, specifies a wide character.
C	Character	When used with `printf` functions, specifies a wide character; when used with `wprintf` functions, specifies a single-byte character.
d	Integer	Signed decimal integer.
i	Integer	Signed decimal integer.
o	Integer	Unsigned octal integer.
u	Integer	Unsigned decimal integer.
x	Integer	Unsigned hexadecimal integer; uses "abcdef."
X	Integer	Unsigned hexadecimal integer; uses "ABCDEF."
e	Floating-point	Signed value that has the form [-]*d.dddd***e±***dd*[*d*] where *d* is one decimal digit, *dddd* is one or more decimal digits depending on the specified precision, or six by default, and *dd*[*d*] is two or three decimal digits depending on the output format and size of the exponent.
E	Floating-point	Identical to the **e** format except that **E** rather than **e** introduces the exponent.

TYPE CHARACTER	ARGUMENT	OUTPUT FORMAT
f	Floating-point	Signed value that has the form [-]*dddd*.*dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision, or six by default.
F	Floating-point	Identical to the **f** format except that infinity and nan output is capitalized.
g	Floating-point	Signed values are displayed in **f** or **e** format, whichever is more compact for the given value and precision. The **e** format is used only when the exponent of the value is less than -4 or greater than or equal to the *precision* argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Floating-point	Identical to the **g** format, except that **E**, rather than **e**, introduces the exponent (where appropriate).
a	Floating-point	Signed hexadecimal double-precision floating-point value that has the form [-]0x*h.hhhh***p±***dd*, where *h.hhhh* are the hex digits (using lower case letters) of the mantissa, and *dd* are one or more digits for the exponent. The precision specifies the number of digits after the point.
A	Floating-point	Signed hexadecimal double-precision floating-point value that has the form [-]0X*h.hhhh***P±***dd*, where *h.hhhh* are the hex digits (using capital letters) of the mantissa, and *dd* are one or more digits for the exponent. The precision specifies the number of digits after the point.
n	Pointer to integer	Number of characters that are successfully written so far to the stream or buffer. This value is stored in the integer whose address is given as the argument. The size of the integer pointed to can be controlled by an argument size specification prefix. The **n** specifier is disabled by default; for information see the important security note.

TYPE CHARACTER	ARGUMENT	OUTPUT FORMAT
p	Pointer type	Displays the argument as an address in hexadecimal digits.
s	String	When used with `printf` functions, specifies a single-byte or multi-byte character string; when used with `wprintf` functions, specifies a wide-character string. Characters are displayed up to the first null character or until the *precision* value is reached.
S	String	When used with `printf` functions, specifies a wide-character string; when used with `wprintf` functions, specifies a single-byte or multi-byte character string. Characters are displayed up to the first null character or until the *precision* value is reached.
Z	`ANSI_STRING` or `UNICODE_STRING` structure	When the address of an ANSI_STRING or UNICODE_STRING structure is passed as the argument, displays the string contained in the buffer pointed to by the `Buffer` field of the structure. Use a *size* modifier prefix of **w** to specify a `UNICODE_STRING` argument—for example, `%wZ`. The `Length` field of the structure must be set to the length, in bytes, of the string. The `MaximumLength` field of the structure must be set to the length, in bytes, of the buffer. Typically, the **Z** type character is used only in driver debugging functions that use a conversion specification, such as `dbgPrint` and `kdPrint`.

Starting in Visual Studio 2015, if the argument that corresponds to a floating-point conversion specifier (**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**) is infinite, indefinite, or NaN, the formatted output conforms to the C99 standard. This table lists the formatted output:

VALUE	OUTPUT
infinity	`inf`
Quiet NaN	`nan`
Signalling NaN	`nan(snan)`
Indefinite NaN	`nan(ind)`

Any of these values may be prefixed by a sign. If a floating-point *type* conversion specifier character is a capital letter, then the output is also formatted in capital letters. For example, if the format specifier is `%F` instead of `%f`, an infinity is formatted as `INF` instead of `inf`. The `scanf` functions can also parse these

strings, so these values can make a round-trip through `printf` and `scanf` functions.

Before Visual Studio 2015, the CRT used a different, non-standard format for output of infinite, indefinite, and NaN values:

VALUE	OUTPUT
+ infinity	`1.#INF` *random-digits*
- infinity	`-1.#INF` *random-digits*
Indefinite (same as quiet NaN)	*digit* `.#IND` *random-digits*
NaN	*digit* `.#NAN` *random-digits*

Any of these may have been prefixed by a sign, and may have been formatted slightly differently depending on field width and precision, sometimes with unusual effects. For example, `printf("%.2f\n", INFINITY)` would print `1.#J` because the #INF would be "rounded" to a precision of 2 digits.

> **NOTE**
> If the argument that corresponds to `%s` or `%S`, or the `Buffer` field of the argument that corresponds to `%Z`, is a null pointer, "(null)" is displayed.

> **NOTE**
> In all exponential formats, the minimum number of digits of exponent to display is two, using three only if necessary. By using the _set_output_format function, you can set the number of digits displayed to three for backward compatibility with code written for Visual Studio 2013 and before.

> **IMPORTANT**
> Because the `%n` format is inherently insecure, it is disabled by default. If `%n` is encountered in a format string, the invalid parameter handler is invoked, as described in Parameter Validation. To enable `%n` support, see _set_printf_count_output.

Flag directives

The first optional field in a conversion specification contains *flag directives*, zero or more flag characters that specify output justification and control output of signs, blanks, leading zeros, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a conversion specification, and the flag characters can appear in any order.

Flag characters

FLAG	MEANING	DEFAULT
-	Left align the result within the given field width.	Right align.
+	Use a sign (+ or -) to prefix the output value if it is of a signed type.	Sign appears only for negative signed values (-).

FLAG	MEANING	DEFAULT
0	If *width* is prefixed by **0**, leading zeros are added until the minimum width is reached. If both **0** and **-** appear, the **0** is ignored. If **0** is specified for an integer format (**i**, **u**, **x**, **X**, **o**, **d**) and a precision specification is also present—for example, `%04.d`—the **0** is ignored. If **0** is specified for the **a** or **A** floating-point format, leading zeros are prepended to the mantissa, after the `0x` or `0X` prefix.	No padding.
blank (' ')	Use a blank to prefix the output value if it is signed and positive. The blank is ignored if both the blank and + flags appear.	No blank appears.
#	When it's used with the **o**, **x**, or **X** format, the **#** flag uses 0, 0x, or 0X, respectively, to prefix any nonzero output value.	No blank appears.
	When it's used with the **e**, **E**, **f**, **F**, **a** or **A** format, the **#** flag forces the output value to contain a decimal point.	Decimal point appears only if digits follow it.
	When it's used with the **g** or **G** format, the **#** flag forces the output value to contain a decimal point and prevents the truncation of trailing zeros. Ignored when used with **c**, **d**, **i**, **u**, or **s**.	Decimal point appears only if digits follow it. Trailing zeros are truncated.

Width specification

In a conversion specification, the optional width specification field appears after any *flags* characters. The *width* argument is a non-negative decimal integer that controls the minimum number of characters that are output. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values—depending on whether the left-alignment flag (**-**) is specified—until the minimum width is reached. If *width* is prefixed by 0, leading zeros are added to integer or floating-point conversions until the minimum width is reached, except when conversion is to an infinity or NaN.

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if *width* is not given, all characters of the value are output, subject to the *precision* specification.

If the width specification is an asterisk (`*`), an `int` argument from the argument list supplies the value. The *width* argument must precede the value that's being formatted in the argument list, as shown in this example:

```
printf("%0*f", 5, 3); /* 00003 is output */
```

A missing or small *width* value in a conversion specification does not cause the truncation of an output value. If the result of a conversion is wider than the *width* value, the field expands to contain the conversion result.

Precision specification

In a conversion specification, the third optional field is the precision specification. It consists of a period (.) followed by a non-negative decimal integer that, depending on the conversion type, specifies the number of string characters, the number of decimal places, or the number of significant digits to be output.

Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value. If *precision* is specified as 0 and the value to be converted is 0, the result is no characters output, as shown in this example:

```
printf( "%.0d", 0 ); /* No characters output */
```

If the precision specification is an asterisk (*), an `int` argument from the argument list supplies the value. In the argument list, the *precision* argument must precede the value that's being formatted, as shown in this example:

```
printf( "%.*f", 3, 3.14159265 ); /* 3.142 output */
```

The *type* character determines either the interpretation of *precision* or the default precision when *precision* is omitted, as shown in the following table.

How Precision Values Affect Type

TYPE	MEANING	DEFAULT
a, **A**	The precision specifies the number of digits after the point.	Default precision is 13. If precision is 0, no decimal point is printed unless the **#** flag is used.
c, **C**	The precision has no effect.	Character is printed.
d, **i**, **o**, **u**, **x**, **X**	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default precision is 1.
e, **E**	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6. If *precision* is 0 or the period (.) appears without a number following it, no decimal point is printed.
f, **F**	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6. If *precision* is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, **G**	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, and any trailing zeros are truncated.
s, **S**	The precision specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed.	Characters are printed until a null character is encountered.

Argument size specification

In a conversion specification, the *size* field is an argument length modifier for the *type* conversion specifier. The *size* field prefixes to the *type* field—**hh**, **h**, **j**, **l** (lowercase L), **L**, **ll**, **t**, **w**, **z**, **I** (uppercase i), **I32**, and **I64**—specify the "size" of the corresponding argument—long or short, 32-bit or 64-bit, single-byte character or wide character—depending on the conversion specifier that they modify. These size prefixes are used with *type* characters in the `printf` and `wprintf` families of functions to specify the interpretation of argument sizes, as shown in the following table. The *size* field is optional for some argument types. When no size prefix is specified, the formatter consumes integer arguments—for example, signed or unsigned `char`, `short`, `int`, `long`, and enumeration types—as 32-bit `int` types, and `float`, `double`, and `long double` floating-point arguments are consumed as 64-bit `double` types. This matches the default argument promotion rules for variable argument lists. For more information about argument promotion, see Ellipses and Default Arguments in Postfix expressions. On both 32-bit and 64-bit systems, the conversion specification of a 64-bit integer argument must include a size prefix of **ll** or **I64**. Otherwise, the behavior of the formatter is undefined.

Some types are different sizes in 32-bit and 64-bit code. For example, `size_t` is 32 bits long in code compiled for x86, and 64 bits in code compiled for x64. To create platform-agnostic formatting code for variable-width types, you can use a variable-width argument size modifier. Alternatively, use a 64-bit argument size modifier and explicitly promote the variable-width argument type to 64 bits. The Microsoft-specific **I** (uppercase i) argument size modifier handles variable-width integer arguments, but we recommend the type-specific **j**, **t**, and **z** modifiers for portability.

Size Prefixes for printf and wprintf Format-Type Specifiers

TO SPECIFY	USE PREFIX	WITH TYPE SPECIFIER
`char` `unsigned char`	**hh**	**d**, **i**, **o**, **u**, **x**, or **X**
`short int` `short unsigned int`	**h**	**d**, **i**, **o**, **u**, **x**, or **X**
`__int32` `unsigned __int32`	**I32**	**d**, **i**, **o**, **u**, **x**, or **X**
`__int64` `unsigned __int64`	**I64**	**d**, **i**, **o**, **u**, **x**, or **X**
`intmax_t` `uintmax_t`	**j** or **I64**	**d**, **i**, **o**, **u**, **x**, or **X**
`long double`	**l** (lowercase L) or **L**	**a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G**
`long int` `long unsigned int`	**l** (lowercase L)	**d**, **i**, **o**, **u**, **x**, or **X**
`long long int` `unsigned long long int`	**ll** (lowercase LL)	**d**, **i**, **o**, **u**, **x**, or **X**
`ptrdiff_t`	**t** or **I** (uppercase i)	**d**, **i**, **o**, **u**, **x**, or **X**
`size_t`	**z** or **I** (uppercase i)	**d**, **i**, **o**, **u**, **x**, or **X**

TO SPECIFY	USE PREFIX	WITH TYPE SPECIFIER
Single-byte character	**h**	**c** or **C**
Wide character	**l** (lowercase L) or **w**	**c** or **C**
Single-byte character string	**h**	**s**, **S**, or **Z**
Wide-character string	**l** (lowercase L) or **w**	**s**, **S**, or **Z**

The `ptrdiff_t` and `size_t` types are `__int32` or `unsigned __int32` on 32-bit platforms, and `__int64` or `unsigned __int64` on 64-bit platforms. The **I** (uppercase i), **j**, **t**, and **z** size prefixes take the correct argument width for the platform.

In Visual C++, although `long double` is a distinct type, it has the same internal representation as `double`.

An **hc** or **hC** type specifier is synonymous with **c** in `printf` functions and with **C** in `wprintf` functions. An **lc**, **lC**, **wc** or **wC** type specifier is synonymous with **C** in `printf` functions and with **c** in `wprintf` functions. An **hs** or **hS** type specifier is synonymous with **s** in `printf` functions and with **S** in `wprintf` functions. An **ls**, **lS**, **ws** or **wS** type specifier is synonymous with **S** in `printf` functions and with **s** in `wprintf` functions.

> **NOTE**
>
> **Microsoft Specific** The **I** (uppercase i), **I32**, **I64**, and **w** argument size modifier prefixes are Microsoft extensions and are not ISO C-compatible. The **h** prefix when it's used with data of type `char` and the **l** (lowercase L) prefix when it's used with data of type `double` are Microsoft extensions.

See also

printf, _printf_l, wprintf, _wprintf_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
printf_p Positional Parameters

Format Specification Fields: scanf and wscanf Functions

3/11/2019 • 3 minutes to read • Edit Online

The information here applies to the entire `scanf` family of functions, including the secure versions and describes the symbols used to tell the `scanf` functions how to parse the input stream, such as the input stream `stdin` for `scanf`, into values that are inserted into program variables.

A format specification has the following form:

`%` [`*`] [width] [{h | l | ll | I64 | L}]type

The `format` argument specifies the interpretation of the input and can contain one or more of the following:

- White-space characters: blank (' '); tab ('\t'); or newline ('\n'). A white-space character causes `scanf` to read, but not store, all consecutive white-space characters in the input up to the next non-white-space character. One white-space character in the format matches any number (including 0) and combination of white-space characters in the input.
- Non-white-space characters, except for the percent sign (`%`). A non-white-space character causes `scanf` to read, but not store, a matching non-white-space character. If the next character in the input stream does not match, `scanf` terminates.
- Format specifications, introduced by the percent sign (`%`). A format specification causes `scanf` to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The format is read from left to right. Characters outside format specifications are expected to match the sequence of characters in the input stream; the matching characters in the input stream are scanned but not stored. If a character in the input stream conflicts with the format specification, `scanf` terminates, and the character is left in the input stream as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to this specification and stored in the location that is specified by the first `argument`. The second format specification causes the second input field to be converted and stored in the second `argument`, and so on through the end of the format string.

An input field is defined as all characters up to the first white-space character (space, tab, or newline), or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached. If there are too many arguments for the given specifications, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments for the format specification.

Each field of the format specification is a single character or a number signifying a particular format option. The `type` character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number.

The simplest format specification contains only the percent sign and a `type` character (for example, `%s`). If a percent sign (`%`) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters, that is, a sequence of characters that must match the input. For example, to specify that a percent-sign character is to be input, use `%%`.

An asterisk (`*`) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The secure versions (those with the `_s` suffix) of the `scanf` family of functions require that a buffer size parameter be passed immediately following each parameter of type `c` , `C` , `s` , `S` or `[` . For more information on the secure versions of the `scanf` family of functions, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l.

See also

scanf Width Specification
scanf Type Field Characters
scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

is, isw Routines

3/11/2019 • 9 minutes to read • Edit Online

isalnum, iswalnum, _isalnum_l, _iswalnum_l	isgraph, iswgraph, _isgraph_l, _iswgraph_l
isalpha, iswalpha, _isalpha_l, _iswalpha_l	isleadbyte, _isleadbyte_l
isascii, __isascii, iswascii	islower, iswlower, _islower_l, _iswlower_l
isblank, iswblank, _isblank_l, _iswblank_l	isprint, iswprint, _isprint_l, _iswprint_l
iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l	ispunct, iswpunct, _ispunct_l, _iswpunct_l
iscsym, iscsymf, __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l	isspace, iswspace, _isspace_l, _iswspace_l
_isctype, iswctype, _isctype_l, _iswctype_l	isupper, _isupper_l, iswupper, _iswupper_l
isdigit, iswdigit, _isdigit_l, _iswdigit_l	isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l

Remarks

These routines test characters for specified conditions.

The **is** routines produce meaningful results for any integer argument from -1 (`EOF`) to **UCHAR_MAX** (0xFF), inclusive. The expected argument type is `int` .

Caution

For the **is** routines, passing an argument of type `char` may yield unpredictable results. An SBCS or MBCS single-byte character of type `char` with a value greater than 0x7F is negative. If a `char` is passed, the compiler may convert the value to a signed `int` or a signed **long**. This value may be sign-extended by the compiler, with unexpected results.

The **isw** routines produce meaningful results for any integer value from - 1 (**WEOF**) to 0xFFFF, inclusive. The **wint_t** data type is defined in WCHAR.H as an **unsigned short**; it can hold any wide character or the wide-character end-of-file (**WEOF**) value.

The output value is affected by the setting of the `LC_CTYPE` category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

In the "C" locale, the test conditions for the **is** routines are as follows:

`isalnum`
Alphanumeric (A - Z, a - z, or 0 - 9).

`isalpha`
Alphabetic (A - Z or a - z).

`__isascii`
ASCII character (0x00 - 0x7F).

`isblank`
Horizontal tab or space character (0x09 or 0x20).

`iscntrl`
Control character (0x00 - 0x1F or 0x7F).

`__iscsym`
Letter, underscore, or digit.

`__iscsymf`
Letter or underscore.

`isdigit`
Decimal digit (0 - 9).

`isgraph`
Printable character except space ( ).

`islower`
Lowercase letter (a - z).

`isprint`
Printable character including space (0x20 - 0x7E).

`ispunct`
Punctuation character.

`isspace`
White-space character (0x09 - 0x0D or 0x20).

`isupper`
Uppercase letter (A - Z).

`isxdigit`
Hexadecimal digit (A - F, a - f, or 0 - 9).

For the **isw** routines, the result of the test for the specified condition is independent of locale. The test conditions for the **isw** functions are as follows:

`iswalnum`
`iswalpha` or `iswdigit`.

`iswalpha`
Any wide character that is one of an implementation-defined set for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is nonzero. `iswalpha` returns nonzero only for wide characters for which `iswupper` or `iswlower` is nonzero.

`iswascii`
Wide-character representation of ASCII character (0x0000 - 0x007F).

`iswblank`
Wide character that corresponds to the standard space character or is one of an implementation-defined set of wide characters for which `iswalnum` is false. Standard blank characters are space (L' ') and horizontal tab (L'\t').

`iswcntrl`
Control wide character.

`__iswcsym`
Any wide character for which `isalnum` is true, or the '_' character.

`__iswcsymf`
Any wide character for which `iswalpha` is true, or the '_' character.

`iswctype`
Character has property specified by the `desc` argument. For each valid value of the `desc` argument of `iswctype`, there is an equivalent wide-character classification routine, as shown in the following table:

Equivalence of iswctype(c, desc) to Other isw Testing Routines

VALUE OF *DESC* ARGUMENT	ISWCTYPE(*C, DESC*) EQUIVALENT
_ALPHA	**iswalpha(`c`)**
_ALPHA | **_DIGIT**	**iswalnum(`c`)**
_BLANK	**iswblank(`c`)**
_CONTROL	**iswcntrl(`c`)**
_DIGIT	**iswdigit(`c`)**
_ALPHA | **_DIGIT** | **_PUNCT**	**iswgraph(`c`)**
_LOWER	**iswlower(`c`)**
_ALPHA | **_BLANK** | **_DIGIT** | **_PUNCT**	**iswprint(`c`)**
_PUNCT	**iswpunct(`c`)**
_BLANK	**iswblank(`c`)**
_SPACE	**iswspace(`c`)**
_UPPER	**iswupper(`c`)**
_HEX	**iswxdigit(`c`)**

`iswdigit`
Wide character corresponding to a decimal-digit character.

`iswgraph`
Printable wide character except space wide character (L' ').

`iswlower`
Lowercase letter, or one of implementation-defined set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is nonzero. `iswlower` returns nonzero only for wide characters that correspond to lowercase letters.

`iswprint`
Printable wide character, including space wide character (L' ').

`iswpunct`
Printable wide character that is neither space wide character (L' ') nor wide character for which `iswalnum` is nonzero.

`iswspace`
Wide character that corresponds to standard white-space character or is one of implementation-defined set of wide characters for which `iswalnum` is false. Standard white-space characters are: space (L' '), formfeed (L'\f'), newline (L'\n'), carriage return (L'\r'), horizontal tab (L'\t'), and vertical tab (L'\v').

`iswupper`
Wide character that is uppercase or is one of an implementation-defined set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is nonzero. `iswupper` returns nonzero only for wide characters that correspond to uppercase characters.

`iswxdigit`
Wide character that corresponds to a hexadecimal-digit character.

Example

```
// crt_isfam.c
/* This program tests all characters between 0x0
* and 0x7F, then displays each character with abbreviations
* for the character-type codes that apply.
*/

#include <stdio.h>
#include <ctype.h>

int main( void )
{
   int ch;
   for( ch = 0; ch <= 0x7F; ch++ )
   {
      printf( "%.2x  ", ch );
      printf( " %c", isprint( ch )  ? ch   : ' ' );
      printf( "%4s", isalnum( ch )  ? "AN" : "" );
      printf( "%3s", isalpha( ch )  ? "A"  : "" );
      printf( "%3s", __isascii( ch )  ? "AS" : "" );
      printf( "%3s", iscntrl( ch )  ? "C"  : "" );
      printf( "%3s", __iscsym( ch )  ? "CS "  : "" );
      printf( "%3s", __iscsymf( ch )  ? "CSF"  : "" );
      printf( "%3s", isdigit( ch )  ? "D"  : "" );
      printf( "%3s", isgraph( ch )  ? "G"  : "" );
      printf( "%3s", islower( ch )  ? "L"  : "" );
      printf( "%3s", ispunct( ch )  ? "PU" : "" );
      printf( "%3s", isspace( ch )  ? "S"  : "" );
      printf( "%3s", isprint( ch )  ? "PR" : "" );
      printf( "%3s", isupper( ch )  ? "U"  : "" );
      printf( "%3s", isxdigit( ch ) ? "X"  : "" );
      printf( ".\n" );
   }
}
```

Output

```
00           AS  C                          .
01           AS  C                          .
02           AS  C                          .
03           AS  C                          .
04           AS  C                          .
05           AS  C                          .
```

06			AS	C									.
07			AS	C									.
08			AS	C									.
09			AS	C						S			.
0a			AS	C						S			.
0b			AS	C						S			.
0c			AS	C						S			.
0d			AS	C						S			.
0e			AS	C									.
0f			AS	C									.
10			AS	C									.
11			AS	C									.
12			AS	C									.
13			AS	C									.
14			AS	C									.
15			AS	C									.
16			AS	C									.
17			AS	C									.
18			AS	C									.
19			AS	C									.
1a			AS	C									.
1b			AS	C									.
1c			AS	C									.
1d			AS	C									.
1e			AS	C									.
1f			AS	C									.
20			AS							S	PR		.
21	!		AS				G		PU		PR		.
22	"		AS				G		PU		PR		.
23	#		AS				G		PU		PR		.
24	$		AS				G		PU		PR		.
25	%		AS				G		PU		PR		.
26	&		AS				G		PU		PR		.
27	'		AS				G		PU		PR		.
28	(		AS				G		PU		PR		.
29	)		AS				G		PU		PR		.
2a	*		AS				G		PU		PR		.
2b	+		AS				G		PU		PR		.
2c	,		AS				G		PU		PR		.
2d	-		AS				G		PU		PR		.
2e	.		AS				G		PU		PR		.
2f	/		AS				G		PU		PR		.
30	0	AN	AS	CS		D	G				PR		X.
31	1	AN	AS	CS		D	G				PR		X.
32	2	AN	AS	CS		D	G				PR		X.
33	3	AN	AS	CS		D	G				PR		X.
34	4	AN	AS	CS		D	G				PR		X.
35	5	AN	AS	CS		D	G				PR		X.
36	6	AN	AS	CS		D	G				PR		X.
37	7	AN	AS	CS		D	G				PR		X.
38	8	AN	AS	CS		D	G				PR		X.
39	9	AN	AS	CS		D	G				PR		X.
3a	:		AS				G		PU		PR		.
3b	;		AS				G		PU		PR		.
3c	<		AS				G		PU		PR		.
3d	=		AS				G		PU		PR		.
3e	>		AS				G		PU		PR		.
3f	?		AS				G		PU		PR		.
40	@		AS				G		PU		PR		.
41	A	AN	A	AS	CS	CSF		G			PR	U	X.
42	B	AN	A	AS	CS	CSF		G			PR	U	X.
43	C	AN	A	AS	CS	CSF		G			PR	U	X.
44	D	AN	A	AS	CS	CSF		G			PR	U	X.
45	E	AN	A	AS	CS	CSF		G			PR	U	X.
46	F	AN	A	AS	CS	CSF		G			PR	U	X.
47	G	AN	A	AS	CS	CSF		G			PR	U	.
48	H	AN	A	AS	CS	CSF		G			PR	U	.
49	I	AN	A	AS	CS	CSF		G			PR	U	.
4a	J	AN	A	AS	CS	CSF		G			PR	U	

```
4a   J  AN  A AS   CS CSF     G          PR  U   .
4b   K  AN  A AS   CS CSF     G          PR  U   .
4c   L  AN  A AS   CS CSF     G          PR  U   .
4d   M  AN  A AS   CS CSF     G          PR  U   .
4e   N  AN  A AS   CS CSF     G          PR  U   .
4f   O  AN  A AS   CS CSF     G          PR  U   .
50   P  AN  A AS   CS CSF     G          PR  U   .
51   Q  AN  A AS   CS CSF     G          PR  U   .
52   R  AN  A AS   CS CSF     G          PR  U   .
53   S  AN  A AS   CS CSF     G          PR  U   .
54   T  AN  A AS   CS CSF     G          PR  U   .
55   U  AN  A AS   CS CSF     G          PR  U   .
56   V  AN  A AS   CS CSF     G          PR  U   .
57   W  AN  A AS   CS CSF     G          PR  U   .
58   X  AN  A AS   CS CSF     G          PR  U   .
59   Y  AN  A AS   CS CSF     G          PR  U   .
5a   Z  AN  A AS   CS CSF     G          PR  U   .
5b   [        AS              G    PU    PR      .
5c   \        AS              G    PU    PR      .
5d   ]        AS              G    PU    PR      .
5e   ^        AS              G    PU    PR      .
5f   _        AS   CS CSF     G    PU    PR      .
60   `        AS              G    PU    PR      .
61   a  AN  A AS   CS CSF     G  L       PR     X.
62   b  AN  A AS   CS CSF     G  L       PR     X.
63   c  AN  A AS   CS CSF     G  L       PR     X.
64   d  AN  A AS   CS CSF     G  L       PR     X.
65   e  AN  A AS   CS CSF     G  L       PR     X.
66   f  AN  A AS   CS CSF     G  L       PR     X.
67   g  AN  A AS   CS CSF     G  L       PR      .
68   h  AN  A AS   CS CSF     G  L       PR      .
69   i  AN  A AS   CS CSF     G  L       PR      .
6a   j  AN  A AS   CS CSF     G  L       PR      .
6b   k  AN  A AS   CS CSF     G  L       PR      .
6c   l  AN  A AS   CS CSF     G  L       PR      .
6d   m  AN  A AS   CS CSF     G  L       PR      .
6e   n  AN  A AS   CS CSF     G  L       PR      .
6f   o  AN  A AS   CS CSF     G  L       PR      .
70   p  AN  A AS   CS CSF     G  L       PR      .
71   q  AN  A AS   CS CSF     G  L       PR      .
72   r  AN  A AS   CS CSF     G  L       PR      .
73   s  AN  A AS   CS CSF     G  L       PR      .
74   t  AN  A AS   CS CSF     G  L       PR      .
75   u  AN  A AS   CS CSF     G  L       PR      .
76   v  AN  A AS   CS CSF     G  L       PR      .
77   w  AN  A AS   CS CSF     G  L       PR      .
78   x  AN  A AS   CS CSF     G  L       PR      .
79   y  AN  A AS   CS CSF     G  L       PR      .
7a   z  AN  A AS   CS CSF     G  L       PR      .
7b   {        AS              G    PU    PR      .
7c   |        AS              G    PU    PR      .
7d   }        AS              G    PU    PR      .
7e   ~        AS              G    PU    PR      .
7f             AS  C                              .
```

See also

Character Classification
Locale
setlocale, _wsetlocale
Interpretation of Multibyte-Character Sequences
to Functions

_ismbb Routines

3/11/2019 • 2 minutes to read • Edit Online

Tests the given integer value `c` for a particular condition, by using the current locale or a specified LC_CTYPE conversion state category.

_ismbbalnum, _ismbbalnum_l	_ismbbkprint, _ismbbkprint_l
_ismbbalpha, _ismbbalpha_l	_ismbbkpunct, _ismbbkpunct_l
_ismbbblank, _ismbbblank_l	_ismbblead, _ismbblead_l
_ismbbgraph, _ismbbgraph_l	_ismbbprint, _ismbbprint_l
_ismbbkalnum, _ismbbkalnum_l	_ismbbpunct, _ismbbpunct_l
_ismbbkana, _ismbbkana_l	_ismbbtrail, _ismbbtrail_l

Remarks

Every routine in the `_ismbb` family tests the given integer value `c` for a particular condition. The test result depends on the multibyte code page that's in effect. By default, the multibyte code page is set to the ANSI code page that's obtained from the operating system at program startup. You can use _getmbcp to query for the multibyte code page that's in use, or _setmbcp to change it.

The output value is affected by the setting of the `LC_CTYPE` category setting of the locale; for more information, see setlocale, _wsetlocale. The versions of these functions that don't have the **_l** suffix use the current locale for this locale-dependent behavior; the versions that do have the **_l** suffix are identical except that instead they use the locale parameter that's passed in.

The routines in the `_ismbb` family test the given integer `c` as follows.

ROUTINE	BYTE TEST CONDITION
_ismbbalnum	`isalnum` || `_ismbbkalnum`.
_ismbbalpha	`isalpha` || `_ismbbkalnum`.
_ismbbblank	`isblank`
_ismbbgraph	Same as `_ismbbprint`, but `_ismbbgraph` does not include the space character (0x20).
_ismbbkalnum	Non-ASCII text symbol other than punctuation. For example, in code page 932 only, `_ismbbkalnum` tests for katakana alphanumeric.
_ismbbkana	Katakana (0xA1 - 0xDF). Specific to code page 932.

ROUTINE	BYTE TEST CONDITION
_ismbbkprint	Non-ASCII text or non-ASCII punctuation symbol. For example, in code page 932 only, `_ismbbkprint` tests for katakana alphanumeric or katakana punctuation (range: 0xA1 - 0xDF).
_ismbbkpunct	Non-ASCII punctuation. For example, in code page 932 only, `_ismbbkpunct` tests for katakana punctuation.
_ismbblead	First byte of multibyte character. For example, in code page 932 only, valid ranges are 0x81 - 0x9F, 0xE0 - 0xFC.
_ismbbprint	`isprint` || `_ismbbkprint`. **ismbbprint** includes the space character (0x20).
_ismbbpunct	`ispunct` || `_ismbbkpunct`.
_ismbbtrail	Second byte of multibyte character. For example, in code page 932 only, valid ranges are 0x40 - 0x7E, 0x80 - 0xEC.

The following table shows the ORed values that compose the test conditions for these routines. The manifest constants `_BLANK`, `_DIGIT`, `_LOWER`, `_PUNCT`, and `_UPPER` are defined in Ctype.h.

ROUTINE	_BLANK	_DIGIT	LOWER	_PUNCT	UPPER	NON-ASCII TEXT	NON-ASCII PUNCT
`_ismbbalnum`	—	x	x	—	x	x	—
`_ismbbalpha`	—	—	x	—	x	x	—
`_ismbbblank`	x	—	—	—	—	—	—
`_ismbbgraph`	—	x	x	x	x	x	x
`_ismbbkalnum`	—	—	—	—	—	x	—
`_ismbbkprint`	—	—	—	—	—	x	x
`_ismbbkpunct`	—	—	—	—	—	—	x
`_ismbbprint`	x	x	x	x	x	x	x
`_ismbbpunct`	—	—	—	x	—	—	x

The `_ismbb` routines are implemented both as functions and as macros. For more information about how to choose either implementation, see Recommendations for Choosing Between Functions and Macros.

See also

Byte Classification

is, isw Routines
_mbbtombc, _mbbtombc_l
_mbctombb, _mbctombb_l

_ismbc Routines

3/11/2019 • 3 minutes to read • Edit Online

Each **_ismbc** routine tests a given multibyte character `c` for a particular condition.

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l	_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l
_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l	_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l
_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l	_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

Remarks

The test result of each **_ismbc** routine depends on the multibyte code page in effect. Multibyte code pages have single-byte alphabetic characters. By default, the multibyte code page is set to the system-default ANSI code page obtained from the operating system at program startup. You can query or change the multibyte code page in use with _getmbcp or _setmbcp, respectively.

The output value is affected by the `LC_CTYPE` category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbcalnum, _ismbcalnum_l	Alphanumeric	Returns nonzero if and only if `c` is a single-byte representation of an ASCII English letter: See examples for `_ismbcdigit` and `_ismbcalpha`.
_ismbcalpha, _ismbcalpha_l	Alphabetic	Returns nonzero if and only if `c` is a single-byte representation of an ASCII English letter: See examples for `_ismbcupper` and `_ismbclower`; or a katakana letter: 0xA6<=`c`<=0xDF.
_ismbcdigit, _ismbcdigit_l	Digit	Returns nonzero if and only if `c` is a single-byte representation of an ASCII digit: 0x30<=`c`<=0x39.
_ismbcgraph, _ismbcgraph_l	Graphic	Returns nonzero if and only if `c` is a single-byte representation of any ASCII or katakana printable character except a white space ( ). See examples for `_ismbcdigit`, `_ismbcalpha`, and `_ismbcpunct`.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbclegal, _ismbclegal_l	Valid multibyte character	Returns nonzero if and only if the first byte of `c` is within ranges 0x81 - 0x9F or 0xE0 - 0xFC, while the second byte is within ranges 0x40 - 0x7E or 0x80 - FC.
_ismbclower, _ismbclower_l	Lowercase alphabetic	Returns nonzero if and only if `c` is a single-byte representation of an ASCII lowercase English letter: 0x61<=`c`<=0x7A.
_ismbcprint, _ismbcprint_l	Printable	Returns nonzero if and only if `c` is a single-byte representation of any ASCII or katakana printable character including a white space ( ): See examples for `_ismbcspace`, `_ismbcdigit`, `_ismbcalpha`, and `_ismbcpunct`.
_ismbcpunct, _ismbcpunct_l	Punctuation	Returns nonzero if and only if `c` is a single-byte representation of any ASCII or katakana punctuation character.
_ismbcblank, _ismbcblank_l,	Space or horizontal tab	Returns nonzero if and only if `c` is a single-byte representation of a space character or a horizontal tab character: `c`=0x20 or `c`=0x09.
_ismbcspace, _ismbcspace_l	Whitespace	Returns nonzero if and only if `c` is a white space character: `c`=0x20 or 0x09<=`c`<=0x0D.
_ismbcsymbol, _ismbcsymbol_l	Multibyte symbol	Returns nonzero if and only if 0x8141<=`c`<=0x81AC.
_ismbcupper, _ismbcupper_l	Uppercase alphabetic	Returns nonzero if and only if `c` is a single-byte representation of an ASCII uppercase English letter: 0x41<=`c`<=0x5A.

Code Page 932 Specific

The following routines are specific to code page 932.

ROUTINE	TEST CONDITION (CODE PAGE 932 ONLY)
_ismbchira, _ismbchira_l	Double-byte Hiragana: 0x829F<=`c`<=0x82F1.
_ismbckata, _ismbckata_l	Double-byte katakana: 0x8340<=`c`<=0x8396.
_ismbcl0, _ismbcl0_l	JIS non-Kanji: 0x8140<=`c`<=0x889E.
_ismbcl1, _ismbcl1_l	JIS level-1: 0x889F<=`c`<=0x9872.

ROUTINE	TEST CONDITION (CODE PAGE 932 ONLY)
_ismbcl2, _ismbcl2_l	JIS level-2: 0x989F<= `c` <=0xEA9E.

`_ismbcl0`, `_ismbcl1`, and `_ismbcl2` check that the specified value `c` matches the test conditions described in the preceding table, but do not check that `c` is a valid multibyte character. If the lower byte is in the ranges 0x00 - 0x3F, 0x7F, or 0xFD - 0xFF, these functions return a nonzero value, indicating that the character satisfies the test condition. Use _ismbbtrail, _ismbbtrail_l to test whether the multibyte character is defined.

END Code Page 932 Specific

See also

Character Classification
is, isw Routines
_ismbb Routines

operator new(CRT)

10/31/2018 • 2 minutes to read • Edit Online

Beginning in Visual Studio 2013, the Universal C Runtime (UCRT) no longer supports the C++-specific operator new and operator delete functions. These are now part of the C++ Standard Library. For more information, see new and delete operators and new operator in the C++ Language Reference.

operator new (CRT)

3/11/2019 • 2 minutes to read • Edit Online

Beginning in Visual Studio 2013, the Universal C Runtime (UCRT) no longer supports the C++-specific operator new and operator delete functions. These are now part of the C++ Standard Library. For more information, see new and delete operators and new operator in the C++ Language Reference.

operator delete(CRT)

10/31/2018 • 2 minutes to read • Edit Online

Beginning in Visual Studio 2013, the Universal C Runtime (UCRT) no longer supports the C++-specific operator new and operator delete functions. These are now part of the C++ Standard Library. For more information, see new and delete operators and delete operator in the C++ Language Reference.

operator delete (CRT)

3/11/2019 • 2 minutes to read • Edit Online

Beginning in Visual Studio 2013, the Universal C Runtime (UCRT) no longer supports the C++-specific operator new and operator delete functions. These are now part of the C++ Standard Library. For more information, see new and delete operators and delete operator in the C++ Language Reference.

printf_p Positional Parameters

3/11/2019 • 2 minutes to read • Edit Online

Positional parameters provide the ability to specify by number which of the arguments is to be substituted into a field in a format string. The following positional parameter `printf` functions are available:

NON-POSITIONAL PRINTF FUNCTIONS	POSITIONAL PARAMETER EQUIVALENTS
printf, _printf_l, wprintf, _wprintf_l	_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l	_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l	_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l	_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
vprintf, _vprintf_l, vwprintf, _vwprintf_l	_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l
vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l	_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l	_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

How to specify positional parameters

Parameter indexing

By default, if no positional formatting is present, the positional functions behave identically to the non-positional ones. You specify the positional parameter to format by using `%n$` at the beginning of the format specifier, where `n` is the position of the parameter to format in the parameter list. The parameter position starts at 1 for the first argument after the format string. The remainder of the format specifier follows the same rules as the `printf` format specifier. For more information about format specfiers, see Format Specification Syntax: printf and wprintf Functions.

Here's an example of positional formatting:

```
_printf_p("%1$s %2$s", "November", "10");
```

This prints:

```
November 10
```

The order of the numbers used doesn't need to match the order of the arguments. For example, this is a valid format string:

```
_printf_p("%2$s %1$s", "November", "10");
```

This prints:

```
10 November
```

Unlike traditional format strings, positional parameters may be used more than once in a format string. For example,

```
_printf_p("%1$d times %1$d is %2$d", 10, 100);
```

This prints:

```
10 times 10 is 100
```

All arguments must be used at least once somewhere in the format string. The maximum number of positional parameters allowed in a format string is given by `_ARGMAX`.

Width and precision

You can use `*n$` to specify a positional parameter as a width or precision specifier, where `n` is the position of the width or precision parameter in the parameter list. The position of the width or precision value must appear immediately following the * symbol. For example,

```
_printf_p("%1$*2$s","Hello", 10);
```

or

```
_printf_p("%2$*1$s", 10, "Hello");
```

Mixing positional and non-positional arguments

Positional parameters may not be mixed with non-positional parameters in the same format string. If any positional formatting is used, all format specifiers must use positional formatting. However, `printf_p` and related functions still support non-positional parameters in format strings containing no positional parameters.

Example

```
// positional_args.c
// Build by using: cl /W4 positional_args.c
// Positional arguments allow the specification of the order
// in which arguments are consumed in a formatting string.

#include <stdio.h>

int main()
{
    int     i = 1,
            j = 2,
            k = 3;
    double  x = 0.1,
            y = 2.22,
            z = 333.3333;
    char    *s1 = "abc",
            *s2 = "def",
            *s3 = "ghi";

    // If positional arguments are unspecified,
    // normal input order is used.
    _printf_p("%d %d %d\n", i, j, k);

    // Positional arguments are numbers followed by a $ character.
    _printf_p("%3$d %1$d %2$d\n", i, j, k);

    // The same positional argument may be reused.
    _printf_p("%1$d %2$d %1$d\n", i, j);

    // The positional arguments may appear in any order.
    _printf_p("%1$s %2$s %3$s\n", s1, s2, s3);
    _printf_p("%3$s %1$s %2$s\n", s1, s2, s3);

    // Precision and width specifiers must be int types.
    _printf_p("%3$*5$f %2$.*4$f %1$*4$.*5$f\n", x, y, z, j, k);
}
```

```
1 2 3
3 1 2
1 2 1
abc def ghi
ghi abc def
333.333300 2.22 0.100
```

See also

Format Specification Syntax: printf and wprintf Functions

scanf Type Field Characters

3/11/2019 • 3 minutes to read • Edit Online

The following information applies to any of the `scanf` family of functions, including the secure versions, such as `scanf_s`.

The `type` character is the only required format field; it appears after any optional format fields. The `type` character determines whether the associated argument is interpreted as a character, string, or number.

Type Characters for scanf functions

CHARACTER	TYPE OF INPUT EXPECTED	TYPE OF ARGUMENT	SIZE ARGUMENT IN SECURE VERSION?
`c`	Character. When used with `scanf` functions, specifies single-byte character; when used with `wscanf` functions, specifies wide character. White-space characters that are ordinarily skipped are read when `c` is specified. To read next non-white-space single-byte character, use `%1s`; to read next non-white-space wide character, use `%1ws`.	Pointer to `char` when used with `scanf` functions, pointer to `wchar_t` when used with `wscanf` functions.	Required. Size does not include space for a null terminator.
`C`	Opposite size character. When used with `scanf` functions, specifies wide character; when used with `wscanf` functions, specifies single-byte character. White-space characters that are ordinarily skipped are read when `C` is specified. To read next non-white-space single-byte character, use `%1s`; to read next non-white-space wide character, use `%1ws`.	Pointer to `wchar_t` when used with `scanf` functions, pointer to `char` when used with `wscanf` functions.	Required. Size argument does not include space for a null terminator.
`d`	Decimal integer.	Pointer to `int`.	No.
`i`	An integer. Hexadecimal if the input string begins with "0x" or "0X", octal if the string begins with "0", otherwise decimal.	Pointer to `int`.	No.
`o`	Octal integer.	Pointer to `int`.	No.

CHARACTER	TYPE OF INPUT EXPECTED	TYPE OF ARGUMENT	SIZE ARGUMENT IN SECURE VERSION?
`p`	A pointer address in hexadecimal digits. The maximum number of digits read depends on the size of a pointer (32 or 64 bits), which depends on the machine architecture. "0x" or "0X" are accepted as prefixes.	Pointer to `void*`.	No.
`u`	Unsigned decimal integer.	Pointer to `unsigned int`.	No.
`x`	Hexadecimal integer.	Pointer to `int`.	No.
`e`, `E`, `f`, `F`, `g`, `G`	Floating-point value consisting of optional sign (+ or -), series of one or more decimal digits containing decimal point, and optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to `float`.	No.
`a`, `A`	Floating-point value consisting of a series of one or more hexadecimal digits containing an optional decimal point, and an exponent ("p" or "P") followed by a decimal value.	Pointer to `float`.	No.
`n`	No input read from stream or buffer.	Pointer to `int`, into which is stored number of characters successfully read from stream or buffer up to that point in current call to `scanf` functions or `wscanf` functions.	No.
`s`	String, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets (`[ ]`), as discussed in scanf Width Specification.	When used with `scanf` functions, signifies single-byte character array; when used with `wscanf` functions, signifies wide-character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.

CHARACTER	TYPE OF INPUT EXPECTED	TYPE OF ARGUMENT	SIZE ARGUMENT IN SECURE VERSION?
`S`	Opposite-size character string, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets (`[ ]`), as discussed in scanf Width Specification.	When used with `scanf` functions, signifies wide-character array; when used with `wscanf` functions, signifies single-byte-character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.

The size arguments, if required, should be passed in the parameter list immediately following the argument they apply to. For example, the following code:

```
char string1[11], string2[9];
scanf_s("%10s %8s", string1, 11, string2, 9);
```

reads a string with a maximum length of 10 into `string1`, and a string with a maximum length of 8 into `string2`. The buffer sizes should be at least one more than the width specifications since space must be reserved for the null terminator.

The format string can handle single-byte or wide character input regardless of whether the single-byte character or wide-character version of the function is used. Thus, to read single-byte or wide characters with `scanf` functions and `wscanf` functions, use format specifiers as follows.

TO READ CHARACTER AS	USE THIS FUNCTION	WITH THESE FORMAT SPECIFIERS
single byte	`scanf` functions	`c`, `hc`, or `hC`
single byte	`wscanf` functions	`C`, `hc`, or `hC`
wide	`wscanf` functions	`c`, `lc`, or `lC`
wide	`scanf` functions	`C`, `lc`, or `lC`

To scan strings with `scanf` functions, and `wscanf` functions, use the above table with format type-specifiers `s` and `S` instead of `c` and `C`.

See also

scanf, _scanf_l, wscanf, _wscanf_l

scanf Width Specification

3/11/2019 • 5 minutes to read • Edit Online

This information applies to the interpretation of format strings in the `scanf` family of functions, including the secure versions such as `scanf_s`. These functions normally assume the input stream is divided into a sequence of tokens. Tokens are separated by whitespace (space, tab, or newline), or in the case of numerical types, by the natural end of a numerical data type as indicated by the first character that cannot be converted into numerical text. However, the width specification may be used to cause parsing of the input to stop before the natural end of a token.

The *width* specification consists of characters between the `%` and the type field specifier, which may include a positive integer called the *width* field and one or more characters indicating the size of the field, which may also be considered as modifiers of the type of the field, such as an indication of whether the integer type is **short** or **long**. Such characters are referred to as the size prefix.

The Width Field

The *width* field is a positive decimal integer controlling the maximum number of characters to be read for that field. No more than *width* characters are converted and stored at the corresponding `argument`. Fewer than *width* characters may be read if a whitespace character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached.

The width specification is separate and distinct from the buffer size argument required by the secure versions of these functions (i.e., `scanf_s`, `wscanf_s`, etc.). In the following example, the width specification is 20, indicating that up to 20 characters are to be read from the input stream. The buffer length is 21, which includes room for the possible 20 characters plus the null terminator:

```
char str[21];
scanf_s("%20s", str, 21);
```

If the *width* field is not used, `scanf_s` will attempt to read the entire token into the string. If the size specified is not large enough to hold the entire token, nothing will be written to the destination string. If the *width* field is specified, then the first *width* characters in the token will be written to the destination string along with the null terminator.

The Size Prefix

The optional prefixes **h**, **l**, **ll**, **I64**, and **L** indicate the size of the `argument` (long or short, single-byte character or wide character, depending upon the type character that they modify). These format-specification characters are used with type characters in `scanf` or `wscanf` functions to specify interpretation of arguments as shown in the following table. The type prefix **I64** is a Microsoft extension and is not ANSI compatible. The type characters and their meanings are described in the "Type Characters for scanf functions" table in scanf Type Field Characters.

> **NOTE**
> The **h**, **l**, and **L** prefixes are Microsoft extensions when used with data of type `char`.

Size Prefixes for scanf and wscanf Format-Type Specifiers

TO SPECIFY	USE PREFIX	WITH TYPE SPECIFIER
double	**l**	**e**, **E**, **f**, **g**, or **G**
long double (same as double)	**L**	**e**, **E**, **f**, **g**, or **G**
long int	**l**	**d**, **i**, **o**, **x**, or **X**
long unsigned int	**l**	**u**
long long	**ll**	**d**, **i**, **o**, **x**, or **X**
`short int`	**h**	**d**, **i**, **o**, **x**, or **X**
short unsigned int	**h**	**u**
__int64	**I64**	**d**, **i**, **o**, **u**, **x**, or **X**
Single-byte character with `scanf`	**h**	**c** or **C**
Single-byte character with `wscanf`	**h**	**c** or **C**
Wide character with `scanf`	**l**	**c** or **C**
Wide character with `wscanf`	**l**	**c**, or **C**
Single-byte - character string with `scanf`	**h**	**s** or **S**
Single-byte - character string with `wscanf`	**h**	**s** or **S**
Wide-character string with `scanf`	**l**	**s** or **S**
Wide-character string with `wscanf`	**l**	**s** or **S**

The following examples use **h** and **l** with `scanf_s` functions and `wscanf_s` functions:

```
scanf_s("%ls", &x, 2);     // Read a wide-character string
wscanf_s(L"%hC", &x, 2);    // Read a single-byte character
```

If using an unsecure function in the `scanf` family, omit the size parameter indicating the buffer length of the preceding argument.

Reading Undelimited strings

To read strings not delimited by whitespace characters, a set of characters in brackets (**[ ]**) can be substituted for the **s** (string) type character. The set of characters in brackets is referred to as a control string. The corresponding input field is read up to the first character that does not appear in the control string. If the first character in the set is a caret (**^**), the effect is reversed: The input field is read up to the first character that does appear in the rest of the character set.

Note that **%[a-z]** and **%[z-a]** are interpreted as equivalent to **%[abcde...z]**. This is a common `scanf` function extension, but note that the ANSI standard does not require it.

Reading Unterminated strings

To store a string without storing a terminating null character ('\0'), use the specification **%*n*c** where *n* is a decimal integer. In this case, the **c** type character indicates that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no null character ('\0') is appended. If *n* is not specified, its default value is 1.

When scanf stops reading a field

The `scanf` function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons:

- The specified width has been reached.
- The next character cannot be converted as specified.
- The next character conflicts with a character in the control string that it is supposed to match.
- The next character fails to appear in a given character set.

For whatever reason, when the `scanf` function stops reading an input field, the next input field is considered to begin at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

See also

scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
Format Specification Fields: scanf and wscanf Functions
scanf Type Field Characters

_spawn, _wspawn Functions

3/11/2019 • 7 minutes to read • Edit Online

Each of the `_spawn` functions creates and executes a new process:

_spawnl, _wspawnl	_spawnv, _wspawnv
_spawnle, _wspawnle	_spawnve, _wspawnve
_spawnlp, _wspawnlp	_spawnvp, _wspawnvp
_spawnlpe, _wspawnlpe	_spawnvpe, _wspawnvpe

The letters at the end of the function name determine the variation.

LETTER	VARIANT
`e`	`envp`, array of pointers to environment settings, is passed to new process.
`l`	Command-line arguments are passed individually to `_spawn` function. This suffix is typically used when a number of parameters to a new process is known in advance.
`p`	`PATH` environment variable is used to find the file to execute.
`v`	`argv`, array of pointers to command-line arguments, is passed to `_spawn` function. This suffix is typically used when a number of parameters to a new process is variable.

Remarks

The `_spawn` functions each create and execute a new process. They automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. The `_wspawn` functions are wide-character versions of the `_spawn` functions; they do not handle multibyte-character strings. Otherwise, the `_wspawn` functions behave identically to their `_spawn` counterparts.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tspawnl`	`_spawnl`	`_spawnl`	`_wspawnl`
`_tspawnle`	`_spawnle`	`_spawnle`	`_wspawnle`

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tspawnlp`	`_spawnlp`	`_spawnlp`	`_wspawnlp`
`_tspawnlpe`	`_spawnlpe`	`_spawnlpe`	`_wspawnlpe`
`_tspawnv`	`_spawnv`	`_spawnv`	`_wspawnv`
`_tspawnve`	`_spawnve`	`_spawnve`	`_wspawnve`
`_tspawnvp`	`_spawnvp`	`_spawnvp`	`_wspawnvp`
`_tspawnvpe`	`_spawnvpe`	`_spawnvpe`	`_wspawnvpe`

Enough memory must be available for loading and executing the new process. The `mode` argument determines the action taken by the calling process before and during `_spawn`. The following values for `mode` are defined in Process.h:

`_P_OVERLAY`	Overlays a calling process with a new process, destroying the calling process (same effect as `_exec` calls).
`_P_WAIT`	Suspends a calling thread until execution of the new process is complete (synchronous `_spawn`).
`_P_NOWAIT` or `_P_NOWAITO`	Continues to execute a calling process concurrently with the new process (asynchronous `_spawn`).
`_P_DETACH`	Continues to execute the calling process; the new process is run in the background with no access to the console or keyboard. Calls to `_cwait` against the new process fail (asynchronous `_spawn`).

The `cmdname` argument specifies the file that is executed as the new process and can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If `cmdname` does not have a file name extension or does not end with a period (.), the `_spawn` function first tries the .com file name extension and then the .exe file name extension, the .bat file name extension, and finally the .cmd file name extension.

If `cmdname` has a file name extension, only that extension is used. If `cmdname` ends with a period, the `_spawn` call searches for `cmdname` with no file name extension. The `_spawnlp`, `_spawnlpe`, `_spawnvp`, and `_spawnvpe` functions search for `cmdname` (using the same procedures) in the directories specified by the `PATH` environment variable.

If `cmdname` contains a drive specifier or any slashes (that is, if it is a relative path), the `_spawn` call searches only for the specified file; no path searching is done.

In the past, some of these functions set `errno` to zero on success; the current behavior is to leave `errno` untouched on success, as specified by the C standard. If you need to emulate the old behavior, set `errno` to zero just before calling these functions.

> **NOTE**
>
> To ensure proper overlay initialization and termination, do not use the `setjmp` or `longjmp` function to enter or leave an overlay routine.

Arguments for the Spawned Process

To pass arguments to the new process, give one or more pointers to character strings as arguments in the `_spawn` call. These character strings form the argument list for the spawned process. The combined length of the strings forming the argument list for the new process must not exceed 1024 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

> **NOTE**
>
> Spaces embedded in strings may cause unexpected behavior; for example, passing `_spawn` the string `"hi there"` will result in the new process getting two arguments, `"hi"` and `"there"`. If the intent was to have the new process open a file named "hi there", the process would fail. You can avoid this by quoting the string: `"\"hi there\""`.

> **IMPORTANT**
>
> Do not pass user input to `_spawn` without explicitly checking its content. `_spawn` will result in a call to CreateProcess so keep in mind that unqualified path names could lead to potential security vulnerabilities.

You can pass argument pointers as separate arguments (in `_spawnl`, `_spawnle`, `_spawnlp`, and `_spawnlpe`) or as an array of pointers (in `_spawnv`, `_spawnve`, `_spawnvp`, and `_spawnvpe`). You must pass at least one argument, `arg0` or `argv`[0], to the spawned process. By convention, this argument is the name of the program as you would type it on the command line. A different value does not produce an error.

The `_spawnl`, `_spawnle`, `_spawnlp`, and `_spawnlpe` calls are typically used in cases where the number of arguments is known in advance. The `arg0` argument is usually a pointer to `cmdname`. The arguments `arg1` through `argn` are pointers to the character strings forming the new argument list. Following `argn`, there must be a **NULL** pointer to mark the end of the argument list.

The `_spawnv`, `_spawnve`, `_spawnvp`, and `_spawnvpe` calls are useful when there is a variable number of arguments to the new process. Pointers to the arguments are passed as an array, `argv`. The argument `argv`[0] is usually a pointer to a path in real mode or to the program name in protected mode, and `argv`[1] through `argv`[`n`] are pointers to the character strings forming the new argument list. The argument `argv`[`n` +1] must be a **NULL** pointer to mark the end of the argument list.

Environment of the Spawned Process

Files that are open when a `_spawn` call is made remain open in the new process. In the `_spawnl`, `_spawnlp`, `_spawnv`, and `_spawnvp` calls, the new process inherits the environment of the calling process. You can use the `_spawnle`, `_spawnlpe`, `_spawnve`, and `_spawnvpe` calls to alter the environment for the new process by passing a list of environment settings through the `envp` argument. The argument `envp` is an array of character pointers, each element (except the final element) of which points to a null-terminated string defining an environment variable. Such a string usually has the form `NAME` = `value` where `NAME` is the name of an environment variable and `value` is the string value to which that variable is set. (Note that `value` is not enclosed in double quotation marks.) The final element of the `envp` array should be **NULL**. When `envp` itself is **NULL**, the spawned process inherits the environment settings of the parent process.

The `_spawn` functions can pass all information about open files, including the translation mode, to the new process. This information is passed in real mode through the `C_FILE_INFO` entry in the environment. The startup code normally processes this entry and then deletes it from the environment. However, if a `_spawn` function spawns a non-C process, this entry remains in the environment. Printing the environment shows graphics characters in the definition string for this entry because the environment information is passed in binary form in real mode. It should not have any other effect on normal operations. In protected mode, the environment information is passed in text form and therefore contains no graphics characters.

You must explicitly flush (using `fflush` or `_flushall`) or close any stream before calling a `_spawn` function.

New processes created by calls to `_spawn` routines do not preserve signal settings. Instead, the spawned process resets signal settings to the default.

Redirecting Output

If you are calling `_spawn` from a DLL or a GUI application and want to redirect the output to a pipe, you have two options:

- Use the Win32 API to create a pipe, then call AllocConsole, set the handle values in the startup structure, and call CreateProcess.
- Call _popen, _wpopen which will create a pipe and invoke the app using **cmd.exe /c** (or **command.exe /c**).

Example

```
// crt_spawn.c
// This program accepts a number in the range
// 1-8 from the command line. Based on the number it receives,
// it executes one of the eight different procedures that
// spawn the process named child. For some of these procedures,
// the CHILD.EXE file must be in the same directory; for
// others, it only has to be in the same path.
//

#include <stdio.h>
#include <process.h>

char *my_env[] =
{
   "THIS=environment will be",
   "PASSED=to child.exe by the",
   "_SPAWNLE=and",
   "_SPAWNLPE=and",
   "_SPAWNVE=and",
   "_SPAWNVPE=functions",
   NULL
};

int main( int argc, char *argv[] )
{
   char *args[4];

   // Set up parameters to be sent:
   args[0] = "child";
   args[1] = "spawn??";
   args[2] = "two";
   args[3] = NULL;

   if (argc <= 2)
   {
      printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
```

```
        exit( 1 );
    }

    switch (argv[1][0])   // Based on first letter of argument
    {
    case '1':
       _spawnl( _P_WAIT, argv[2], argv[2], "_spawnl", "two", NULL );
       break;
    case '2':
       _spawnle( _P_WAIT, argv[2], argv[2], "_spawnle", "two",
                NULL, my_env );
       break;
    case '3':
       _spawnlp( _P_WAIT, argv[2], argv[2], "_spawnlp", "two", NULL );
       break;
    case '4':
       _spawnlpe( _P_WAIT, argv[2], argv[2], "_spawnlpe", "two",
                 NULL, my_env );
       break;
    case '5':
       _spawnv( _P_OVERLAY, argv[2], args );
       break;
    case '6':
       _spawnve( _P_OVERLAY, argv[2], args, my_env );
       break;
    case '7':
       _spawnvp( _P_OVERLAY, argv[2], args );
       break;
    case '8':
       _spawnvpe( _P_OVERLAY, argv[2], args, my_env );
       break;
    default:
       printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
       exit( 1 );
    }
    printf( "from SPAWN!\n" );
}
```

```
child process output
from SPAWN!
```

See also

Process and Environment Control
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

strcoll Functions

3/11/2019 • 2 minutes to read • Edit Online

Each of the `strcoll` and `wcscoll` functions compares two strings according to the `LC_COLLATE` category setting of the locale code page currently in use. Each of the `_mbscoll` functions compares two strings according to the multibyte code page currently in use. Use the `coll` functions for string comparisons when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the comparison. Use the corresponding `cmp` functions to test only for string equality.

strcoll Functions

SBCS	UNICODE	MBCS	DESCRIPTION
strcoll	wcscoll	_mbscoll	Collate two strings
_stricoll	_wcsicoll	_mbsicoll	Collate two strings (case insensitive)
_strncoll	_wcsncoll	_mbsncoll	Collate first `count` characters of two strings
_strnicoll	_wcsnicoll	_mbsnicoll	Collate first `count` characters of two strings (case-insensitive)

Remarks

The single-byte character (SBCS) versions of these functions (`strcoll` , `stricoll` , `_strncoll` , and `_strnicoll`) compare `string1` and `string2` according to the `LC_COLLATE` category setting of the current locale. These functions differ from the corresponding `strcmp` functions in that the `strcoll` functions use locale code page information that provides collating sequences. For string comparisons in locales in which the character set order and the lexicographic character order differ, the `strcoll` functions should be used rather than the corresponding `strcmp` functions. For more information on `LC_COLLATE` , see setlocale.

For some code pages and corresponding character sets, the order of characters in the character set may differ from the lexicographic character order. In the "C" locale, this is not the case: the order of characters in the ASCII character set is the same as the lexicographic order of the characters. However, in certain European code pages, for example, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically. To perform a lexicographic comparison in such an instance, use `strcoll` rather than `strcmp` . Alternatively, you can use `strxfrm` on the original strings, then use `strcmp` on the resulting strings.

`strcoll` , `stricoll` , `_strncoll` , and `_strnicoll` automatically handle multibyte-character strings according to the locale code page currently in use, as do their wide-character (Unicode) counterparts. The multibyte-character (MBCS) versions of these functions, however, collate strings on a character basis according to the multibyte code page currently in use.

Because the `coll` functions collate strings lexicographically for comparison, whereas the `cmp` functions simply test for string equality, the `coll` functions are much slower than the corresponding `cmp` versions. Therefore, the `coll` functions should be used only when there is a difference between the character set order and the

lexicographic character order in the current code page and this difference is of interest for the string comparison.

See also

Locale
String Manipulation
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

String to Numeric Value Functions

3/11/2019 • 3 minutes to read • Edit Online

- strtod, _strtod_l, wcstod, _wcstod_l
- strtol, wcstol, _strtol_l, _wcstol_l
- strtoul, _strtoul_l, wcstoul, _wcstoul_l
- _strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l
- _strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

Remarks

Each function in the **strtod** family converts a null-terminated string to a numeric value. The available functions are listed in the following table.

FUNCTION	DESCRIPTION
`strtod`	Convert string to double-precision floating point value
`strtol`	Convert string to long integer
`strtoul`	Convert string to unsigned long integer
`_strtoi64`	Convert string to 64-bit `__int64` integer
`_strtoui64`	Convert string to unsigned 64-bit `__int64` integer

`wcstod`, `wcstol`, `wcstoul`, and `_wcstoi64` are wide-character versions of `strtod`, `strtol`, `strtoul`, and `_strtoi64`, respectively. The string argument to each of these wide-character functions is a wide-character string; each function behaves identically to its single-byte-character counterpart otherwise.

The `strtod` function takes two arguments: the first is the input string, and the second a pointer to the character which ends the conversion process. `strtol`, `strtoul`, **_strtoi64** and **_strtoui64** take a third argument as the number base to use in the conversion process.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. Each function stops reading the string at the first character it cannot recognize as part of a number. This may be the terminating null character. For `strtol`, `strtoul`, `_strtoi64`, and `_strtoui64`, this terminating character can also be the first numeric character greater than or equal to the user-supplied number base.

If the user-supplied pointer to an end-of-conversion character is not set to **NULL** at call time, a pointer to the character that stopped the scan will be stored there instead. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of the string pointer is stored at that address.

`strtod` expects a string of the following form:

[*whitespace*] [*sign*] [`digits`] [**.** `digits`] [{**d** | **D** | **e** | **E**}[*sign*] `digits`]

A *whitespace* may consist of space or tab characters, which are ignored; *sign* is either plus (**+**) or minus (**-**); and `digits` are one or more decimal digits. If no digits appear before the radix character, at least one must appear

after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

The `strtol`, `strtoul`, `_strtoi64`, and `_strtoui64` functions expect a string of the following form:

[*whitespace*] [{**+** | **-**}] [**0** [{ **x** | **X** }]] [`digits`]

If the base argument is between 2 and 36, then it is used as the base of the number. If it is 0, the initial characters referenced to by the end-of-conversion pointer are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. `strtoul` and `_strtoui64` allow a plus (**+**) or minus (**-**) sign prefix; a leading minus sign indicates that the return value is negated.

The output value is affected by the setting of the `LC_NUMERIC` category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead.

When the value returned by these functions would cause an overflow or underflow, or when conversion is not possible, special case values are returned as shown:

FUNCTION	CONDITION	VALUE RETURNED
`strtod`	Overflow	+/- `HUGE_VAL`
`strtod`	Underflow or no conversion	0
`strtol`	+ Overflow	**LONG_MAX**
`strtol`	- Overflow	**LONG_MIN**
`strtol`	Underflow or no conversion	0
`_strtoi64`	+ Overflow	**_I64_MAX**
`_strtoi64`	- Overflow	**_I64_MIN**
`_strtoi64`	No conversion	0
`_strtoui64`	Overflow	**_UI64_MAX**
`_strtoui64`	No conversion	0

_I64_MAX, **_I64_MIN**, and **_UI64_MAX** are defined in LIMITS.H.

`wcstod`, `wcstol`, `wcstoul`, `_wcstoi64`, and `_wcstoui64` are wide-character versions of `strtod`, `strtol`, `strtoul`, `_strtoi64`, and `_strtoui64`, respectively; the pointer to an end-of-conversion argument to each of these wide-character functions is a wide-character string. Otherwise, each of these wide-character functions behaves identically to its single-byte-character counterpart.

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l

to Functions

3/11/2019 • 2 minutes to read • Edit Online

Each of the **to** functions and its associated macro, if any, converts a single character to another character.

__toascii	toupper, _toupper, towupper
tolower, _tolower, towlower	

Remarks

The **to** functions and macro conversions are as follows.

ROUTINE	MACRO	DESCRIPTION
`__toascii`	`__toascii`	Converts `c` to ASCII character
`tolower`	`tolower`	Converts `c` to lowercase if appropriate
`_tolower`	`_tolower`	Converts `c` to lowercase
`towlower`	None	Converts `c` to corresponding wide-character lowercase letter
`toupper`	`toupper`	Converts `c` to uppercase if appropriate
`_toupper`	`_toupper`	Converts `c` to uppercase
`towupper`	None	Converts c to corresponding wide-character uppercase letter

To use the function versions of the **to** routines that are also defined as macros, either remove the macro definitions with `#undef` directives or do not include CTYPE.H. If you use the /Za compiler option, the compiler uses the function version of `toupper` or `tolower`. Declarations of the `toupper` and `tolower` functions are in STDLIB.H.

The `__toascii` routine sets all but the low-order 7 bits of `c` to 0, so that the converted value represents a character in the ASCII character set. If `c` already represents an ASCII character, `c` is unchanged.

The `tolower` and `toupper` routines:

- Are dependent on the `LC_CTYPE` category of the current locale (`tolower` calls `isupper` and `toupper` calls `islower`).
- Convert `c` if `c` represents a convertible letter of the appropriate case in the current locale and the opposite case exists for that locale. Otherwise, `c` is unchanged.

The `_tolower` and `_toupper` routines:

- Are locale-independent, much faster versions of `tolower` and **toupper.**
- Can be used only when **isascii(** `c` **)** and either **isupper(** `c` **)** or **islower(** `c` **)**, respectively, are nonzero.
- Have undefined results if `c` is not an ASCII letter of the appropriate case for converting.

The `towlower` and `towupper` functions return a converted copy of `c` if and only if both of the following conditions are nonzero. Otherwise, `c` is unchanged.

- `c` is a wide character of the appropriate case (that is, for which `iswupper` or **iswlower,** respectively, is nonzero).
- There is a corresponding wide character of the target case (that is, for which `iswlower` or **iswupper,** respectively, is nonzero).

Example

```
// crt_toupper.c
/* This program uses toupper and tolower to
 * analyze all characters between 0x0 and 0x7F. It also
 * applies _toupper and _tolower to any code in this
 * range for which these functions make sense.
 */

#include <ctype.h>
#include <string.h>

char msg[] = "Some of THESE letters are Capitals.";
char *p;

int main( void )
{
   printf( "%s\n", msg );

   /* Reverse case of message. */
   for( p = msg; p < msg + strlen( msg ); p++ )
   {
      if( islower( *p ) )
         putchar( _toupper( *p ) );
      else if( isupper( *p ) )
         putchar( _tolower( *p ) );
      else
         putchar( *p );
   }
}
```

```
Some of THESE letters are Capitals.
sOME OF these LETTERS ARE cAPITALS.
```

See also

Data Conversion
Locale
is, isw Routines

vprintf Functions

3/11/2019 • 3 minutes to read • Edit Online

Each of the `vprintf` functions takes a pointer to an argument list, then formats and writes the given data to a particular destination. The functions differ in the parameter validation performed, whether the functions take wide or single-byte character strings, the output destination, and the support for specifying the order in which parameters are used in the format string.

_vcprintf, _vcwprintf	vfprintf, vfwprintf
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l	vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l
vprintf, vwprintf	_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l
vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l	vsprintf, vswprintf
_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l	vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l
_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l	_vsnprintf, _vsnwprintf

Remarks

The `vprintf` functions are similar to their counterpart functions as listed in the following table. However, each `vprintf` function accepts a pointer to an argument list, whereas each of the counterpart functions accepts an argument list.

These functions format data for output to destinations as follows.

FUNCTION	COUNTERPART FUNCTION	OUTPUT DESTINATION	PARAMETER VALIDATION	POSITIONAL PARAMETER SUPPORT
`_vcprintf`	_cprintf	console	Check for null.	no
`_vcwprintf`	_cwprintf	console	Check for null.	no
`vfprintf`	fprintf	*Stream*	Check for null.	no
vfprintf_p	fprintf_p	*Stream*	Check for null and valid format.	yes
`vfprintf_s`	fprintf_s	*Stream*	Check for null and valid format.	no
`vfwprintf`	fwprintf	*Stream*	Check for null.	no
vfwprintf_p	fwprintf_p	*Stream*	Check for null and valid format.	yes

FUNCTION	COUNTERPART FUNCTION	OUTPUT DESTINATION	PARAMETER VALIDATION	POSITIONAL PARAMETER SUPPORT
`vfwprintf_s`	fwprintf_s	*Stream*	Check for null and valid format.	no
`vprintf`	printf	`Stdout`	Check for null.	no
vprintf_p	printf_p	`Stdout`	Check for null and valid format.	yes
`vprintf_s`	printf_s	`Stdout`	Check for null and valid format.	no
`vwprintf`	wprintf	`Stdout`	Check for null.	no
vwprintf_p	wprintf_p	`Stdout`	Check for null and valid format.	yes
`vwprintf_s`	wprintf_s	`Stdout`	Check for null and valid format.	no
vsprintf	sprintf	memory pointed to by *buffer*	Check for null.	no
vsprintf_p	sprintf_p	memory pointed to by *buffer*	Check for null and valid format.	yes
`vsprintf_s`	sprintf_s	memory pointed to by *buffer*	Check for null and valid format.	no
`vswprintf`	swprintf	memory pointed to by *buffer*	Check for null.	no
vswprintf_p	swprintf_p	memory pointed to by *buffer*	Check for null and valid format.	yes
`vswprintf_s`	swprintf_s	memory pointed to by *buffer*	Check for null and valid format.	no
`_vscprintf`	_vscprintf	memory pointed to by *buffer*	Check for null.	no
`_vscwprintf`	_vscwprintf	memory pointed to by *buffer*	Check for null.	no
`_vsnprintf`	_snprintf	memory pointed to by *buffer*	Check for null.	no
`_vsnwprintf`	_snwprintf	memory pointed to by *buffer*	Check for null.	no

The `argptr` argument has type `va_list`, which is defined in VARARGS.H and STDARG.H. The `argptr` variable must be initialized by **va_start,** and may be reinitialized by subsequent `va_arg` calls; `argptr` then points to the beginning of a list of arguments that are converted and transmitted for output according to the corresponding

specifications in the *format* argument. *format* has the same form and function as the *format* argument for printf. None of these functions invokes `va_end`. For a more complete description of each `vprintf` function, see the description of its counterpart function as listed in the preceding table.

`_vsnprintf` differs from **vsprintf** in that it writes no more than *count* bytes to *buffer*.

The versions of these functions with the **w** infix in the name are wide-character versions of the corresponding functions without the **w** infix; in each of these wide-character functions, *buffer* and *format* are wide-character strings. Otherwise, each wide-character function behaves identically to its SBCS counterpart function.

The versions of these functions with **_s** and **_p** suffixes are the more secure versions. These versions validate the format strings and will generate an exception if the format string is not well formed (for example, if invalid formatting characters are used).

The versions of these functions with the **_p** suffix provide the ability to specify the order in which the supplied arguments are substituted in the format string. For more information, see printf_p Positional Parameters.

For **vsprintf**, `vswprintf`, `_vsnprintf` and `_vsnwprintf`, if copying occurs between strings that overlap, the behavior is undefined.

> **IMPORTANT**
>
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns. If using the secure versions of these functions (either the **_s** or **_p** suffixes), a user-supplied format string could trigger an invalid parameter exception if the user-supplied string contains invalid formatting characters.

See also

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

Obsolete Functions

2/4/2019 • 2 minutes to read • Edit Online

Certain library functions are obsolete and have more recent equivalents. We recommend you change these to the updated versions. Other obsolete functions have been removed from the CRT. This topic lists the functions deprecated as obsolete, and the functions removed in a particular version of Visual Studio.

Deprecated as obsolete in Visual Studio 2015

OBSOLETE FUNCTION	ALTERNATIVE
`is_wctype`	iswctype
`_loaddll`	LoadLibrary, LoadLibraryEx, or LoadPackagedLibrary
`_unloaddll`	FreeLibrary
`_getdllprocaddr`	GetProcAddress
`_seterrormode`	SetErrorMode
`_beep`	Beep
`_sleep`	Sleep
`_getsystime`	GetLocalTime
`_setsystime`	SetLocalTime

Removed from the CRT in Visual Studio 2015

OBSOLETE FUNCTION	ALTERNATIVE
_cgets, _cgetws	_cgets_s, _cgetws_s
gets, _getws	gets_s, _getws_s
_get_output_format	None
_heapadd	None
_heapset	None
inp, inpw	None
_inp, _inpw, _inpd	None

OBSOLETE FUNCTION	ALTERNATIVE
outp, outpw	None
_outp, _outpw, _outpd	None
_set_output_format	None

Removed from the CRT in earlier versions of Visual Studio

_lock

_unlock

_cgets, _cgetws

3/11/2019 • 2 minutes to read • Edit Online

Gets a character string from the console. More secure versions of these functions are available; see _cgets_s, _cgetws_s.

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT. The secure versions of these functions, _cgets_s and _cgetws_s, are still available. For information on these alternative functions, see _cgets_s, _cgetws_s.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_cgets(
   char *buffer
);
wchar_t *_cgetws(
   wchar_t *buffer
);
template <size_t size>
char *_cgets(
   char (&buffer)[size]
); // C++ only
template <size_t size>
wchar_t *_cgetws(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer
Storage location for data.

Return Value

`_cgets` and `_cgetws` return a pointer to the start of the string, at `buffer[2]`. If `buffer` is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, they return **NULL** and set `errno` to `EINVAL`.

Remarks

These functions read a string of characters from the console and store the string and its length in the location pointed to by `buffer`. The `buffer` parameter must be a pointer to a character array. The first element of the array, `buffer[0]`, must contain the maximum length (in characters) of the string to be read. The array must contain enough elements to hold the string, a terminating null character ('\0'), and 2 additional bytes. The

function reads characters until a carriage return-line feed (CR-LF) combination or the specified number of characters is read. The string is stored starting at `buffer[2]`. If the function reads a CR-LF, it stores the null character ('\0'). The function then stores the actual length of the string in the second array element, `buffer[1]`.

Because all editing keys are active when `_cgets` or `_cgetws` is called while in a console window, pressing the F3 key repeats the last entered entry.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_cgetts`	`_cgets`	`_cgets`	`_cgetws`

Requirements

ROUTINE	REQUIRED HEADER
`_cgets`	<conio.h>
`_cgetws`	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cgets.c
// compile with: /c /W3
// This program creates a buffer and initializes
// the first byte to the size of the buffer. Next, the
// program accepts an input string using _cgets and displays
// the size and text of that string.

#include <conio.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
   char buffer[83] = { 80 };  // Maximum characters in 1st byte
   char *result;

   printf( "Input line of text, followed by carriage return:\n");

   // Input a line of text:
   result = _cgets( buffer ); // C4996
   // Note: _cgets is deprecated; consider using _cgets_s
   if (!result)
   {
      printf( "An error occurred reading from the console:"
              " error code %d\n", errno);
   }
   else
   {
      printf( "\nLine length = %d\nText = %s\n",
              buffer[1], result );
   }
}
```

```
      A line of input.Input line of text, followed by carriage return:
Line Length = 16
Text = A line of input.
```

See also

Console and Port I/O
_getch, _getwch

_get_output_format

3/11/2019 • 2 minutes to read • Edit Online

Gets the current value of the output format flag.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
unsigned int _get_output_format();
```

Return Value

The current value of the output format flag.

Remarks

The output format flag controls features of formatted I/O. At present the flag has two possible values: 0 and `_TWO_DIGIT_EXPONENT`. If `_TWO_DIGIT_EXPONENT` is set, the floating point numbers is printed with only two digits in the exponent unless a third digit is required by the size of the exponent. If the flag is zero, the floating point output displays three digits of exponent, using zeroes if necessary to pad the value to three digits.

Requirements

ROUTINE	REQUIRED HEADER
`_get_output_format`	<stdio.h>

For more compatibility information, see Compatibility in the Introduction.

See also

Format Specification Syntax: printf and wprintf Functions
printf, _printf_l, wprintf, _wprintf_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
_set_output_format

gets, _getws

3/11/2019 • 2 minutes to read • Edit Online

Gets a line from the `stdin` stream. More secure versions of these functions are available; see gets_s, _getws_s.

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT. The secure versions of these functions, gets_s and _getws_s, are still available. For information on these alternative functions, see gets_s, _getws_s.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *gets(
   char *buffer
);
wchar_t *_getws(
   wchar_t *buffer
);
template <size_t size>
char *gets(
   char (&buffer)[size]
); // C++ only
template <size_t size>
wchar_t *_getws(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer

Storage location for input string.

Return Value

Returns its argument if successful. A **NULL** pointer indicates an error or end-of-file condition. Use ferror or feof to determine which one has occurred. If `buffer` is **NULL**, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set errno to `EINVAL`.

Remarks

The `gets` function reads a line from the standard input stream `stdin` and stores it in `buffer`. The line consists of all characters up to and including the first newline character ('\n'). `gets` then replaces the newline character with a null character ('\0') before returning the line. In contrast, the `fgets` function retains the newline character. `_getws` is a wide-character version of `gets`; its argument and return value are wide-character strings.

> **IMPORTANT**
> Because there is no way to limit the number of characters read by gets, untrusted input can easily cause buffer overruns. Use `fgets` instead.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_getts`	`gets`	`gets`	`_getws`

Requirements

ROUTINE	REQUIRED HEADER
`gets`	<stdio.h>
`_getws`	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_gets.c
// compile with: /WX /W3

#include <stdio.h>

int main( void )
{
   char line[21]; // room for 20 chars + '\0'
   gets( line );  // C4996
   // Danger: No way to limit input to 20 chars.
   // Consider using gets_s instead.
   printf( "The line entered was: %s\n", line );
}
```

Note that input longer than 20 characters will overrun the line buffer and almost certainly cause the program to crash.

```
Hello there!The line entered was: Hello there!
```

See also

Stream I/O
fgets, fgetws
fputs, fputws
puts, _putws

_heapadd

3/11/2019 • 2 minutes to read • Edit Online

Adds memory to the heap.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
int _heapadd(
   void *memblock,
   size_t size
);
```

Parameters

memblock
Pointer to the heap memory.

size
Size of memory to add, in bytes.

Return Value

If successful, `_heapadd` returns 0; otherwise, the function returns -1 and sets `errno` to `ENOSYS`.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Beginning with Visual C++ version 4.0, the underlying heap structure was moved to the C run-time libraries to support the new debugging features. As a result, `_heapadd` is no longer supported on any platform that is based on the Win32 API.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
`_heapadd`	<malloc.h>	<errno.h>

For more compatibility information, see Compatibility in the Introduction.

See also

Memory Allocation
free
_heapchk
_heapmin

_heapset
_heapwalk
malloc
realloc

_heapset

3/11/2019 • 2 minutes to read • Edit Online

Checks heaps for minimal consistency and sets the free entries to a specified value.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
int _heapset(
   unsigned int fill
);
```

Parameters

fill
Fill character.

Return Value

`_heapset` returns one of the following integer manifest constants defined in Malloc.h.

`_HEAPBADBEGIN`	Initial header information invalid or not found.
`_HEAPBADNODE`	Heap damaged or bad node found.
`_HEAPEMPTY`	Heap not initialized.
`_HEAPOK`	Heap appears to be consistent.

In addition, if an error occurs, `_heapset` sets `errno` to `ENOSYS`.

Remarks

The `_heapset` function shows free memory locations or nodes that have been unintentionally overwritten.

`_heapset` checks for minimal consistency on the heap and then sets each byte of the heap's free entries to the `fill` value. This known value shows which memory locations of the heap contain free nodes and which contain data that were unintentionally written to freed memory. If the operating system does not support `_heapset` (for example, Windows 98), the function returns `_HEAPOK` and sets `errno` to `ENOSYS`.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
`_heapset`	<malloc.h>	<errno.h>

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_heapset.c
// This program checks the heap and
// fills in free entries with the character 'Z'.

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int heapstatus;
   char *buffer;

   if( (buffer = malloc( 1 )) == NULL ) // Make sure heap is
      exit( 0 );                        //    initialized
   heapstatus = _heapset( 'Z' );        // Fill in free entries
   switch( heapstatus )
   {
   case _HEAPOK:
      printf( "OK - heap is fine\n" );
      break;
   case _HEAPEMPTY:
      printf( "OK - heap is empty\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
   }
   free( buffer );
}
```

```
OK - heap is fine
```

See also

Memory Allocation
_heapadd
_heapchk
_heapmin
_heapwalk

inp, inpw

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _inp, _inpw, _inpd instead.

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_inp, _inpw, _inpd

3/11/2019 • 2 minutes to read • Edit Online

Inputs, from a port, a byte (`_inp`), a word (`_inpw`), or a double word (`_inpd`).

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _inp(
   unsigned short port
);
unsigned short _inpw(
   unsigned short port
);
unsigned long _inpd(
   unsigned short port
);
```

Parameters

port
I/O port number.

Return Value

The functions return the byte, word, or double word read from `port` . There is no error return.

Remarks

The `_inp` , `_inpw` , and `_inpd` functions read a byte, a word, and a double word, respectively, from the specified input port. The input value can be any unsigned short integer in the range 0 - 65,535.

Because these functions read directly from an I/O port, they cannot be used in user code.

Requirements

ROUTINE	REQUIRED HEADER
`_inp`	<conio.h>
`_inpw`	<conio.h>

ROUTINE	REQUIRED HEADER
`_inpd`	<conio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Console and Port I/O
_outp, _outpw, _outpd

_lock

3/11/2019 • 2 minutes to read • Edit Online

Acquires a multi-thread lock.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
void __cdecl _lock
   int locknum
);
```

Parameters

locknum
[in] The identifier of the lock to acquire.

Remarks

If the lock has already been acquired, this method acquires the lock anyway and causes an internal C run-time (CRT) error. If the method cannot acquire a lock, it exits with a fatal error and sets the error code to `_RT_LOCK`.

Requirements

Source: mlock.c

See also

Alphabetical Function Reference
_unlock

outp, outpw

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _outp, _outpw, _outpd instead.

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_outp, _outpw, _outpd

3/11/2019 • 2 minutes to read • Edit Online

Outputs, at a port, a byte (`_outp`), a word (`_outpw`), or a double word (`_outpd`).

IMPORTANT

These functions are obsolete. Beginning in Visual Studio 2015, they are not available in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
      int _outp(
unsigned short port,
int databyte
);
unsigned short _outpw(
unsigned short port,
unsigned short dataword
);
unsigned long _outpd(
unsigned short port,
unsigned long dataword
);
```

Parameters

port
Port number.

databyte, dataword
Output values.

Return Value

The functions return the data output. There is no error return.

Remarks

The `_outp` , `_outpw` , and `_outpd` functions write a byte, a word, and a double word, respectively, to the specified output port. The *port* argument can be any unsigned integer in the range 0 - 65,535; *databyte* can be any integer in the range 0 - 255; and *dataword* can be any value in the range of an integer, an unsigned short integer, and an unsigned long integer, respectively.

Because these functions write directly to an I/O port, they cannot be used in user code. For information about using I/O ports in these operating systems, search for "Serial Communications in Win32" at MSDN.

Requirements

ROUTINE	REQUIRED HEADER
`_outp`	<conio.h>
`_outpw`	<conio.h>
`_outpd`	<conio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Console and Port I/O
_inp, _inpw, _inpd

_set_output_format

3/11/2019 • 2 minutes to read • Edit Online

Customizes output formats used by formatted I/O functions.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
unsigned int _set_output_format(
   unsigned int format
);
```

Parameters

format
[in] An value representing the format to use.

Return value

The previous output format.

Remarks

`_set_output_format` is used to configure the output of formatted I/O functions such as printf_s. At present, the only formatting convention that can be changed by this function is the number of digits displayed in exponents in the output of floating point numbers.

By default, the output of floating point numbers by functions such as `printf_s`, `wprintf_s`, and related functions in the Visual C++ Standard C library prints three digits for the exponent, even if three digits are not required to represent the value of the exponent. Zeroes are used to pad the value to three digits. `_set_output_format` allows you to change this behavior so that only two digits are printed in the exponent unless a third digit is required by the size of the exponent.

To enable two-digit exponents, call this function with the parameter `_TWO_DIGIT_EXPONENT`, as shown in the example. To disable two digit exponents, call this function with an argument of 0.

Requirements

ROUTINE	REQUIRED HEADER
`_set_output_format`	<stdio.h>

For more compatibility information, see Compatibility in the Introduction.

Example

```
// crt_set_output_format.c
#include <stdio.h>

void printvalues(double x, double y)
{
   printf_s("%11.4e %11.4e\n", x, y);
   printf_s("%11.4E %11.4E\n", x, y);
   printf_s("%11.4g %11.4g\n", x, y);
   printf_s("%11.4G %11.4G\n", x, y);
}

int main()
{
   double x = 1.211E-5;
   double y = 2.3056E-112;
   unsigned int old_exponent_format;

   // Use the default format
   printvalues(x, y);

   // Enable two-digit exponent format
   old_exponent_format = _set_output_format(_TWO_DIGIT_EXPONENT);

   printvalues(x, y);

   // Disable two-digit exponent format
   _set_output_format( old_exponent_format );

   printvalues(x, y);
}
```

```
1.2110e-005 2.3056e-112
1.2110E-005 2.3056E-112
1.211e-005  2.306e-112
1.211E-005  2.306E-112
1.2110e-05 2.3056e-112
1.2110E-05 2.3056E-112
  1.211e-05  2.306e-112
  1.211E-05  2.306E-112
1.2110e-005 2.3056e-112
1.2110E-005 2.3056E-112
1.211e-005  2.306e-112
1.211E-005  2.306E-112
```

See also

printf_s, _printf_s_l, wprintf_s, _wprintf_s_l

_get_output_format

_unlock

3/11/2019 • 2 minutes to read • Edit Online

Releases a multi-thread lock.

> **IMPORTANT**
> This function is obsolete. Beginning in Visual Studio 2015, it is not available in the CRT.

Syntax

```
void __cdecl _unlock(
   int locknum
);
```

Parameters

locknum
[in] The identifier of the lock to release.

Requirements

Source: mlock.c

See also

Alphabetical Function Reference
_lock

UCRT alphabetical function reference

2/4/2019 • 8 minutes to read • Edit Online

The Universal C Runtime (UCRT, often just CRT) Library reference documentation is arranged alphabetically by routine. To find a CRT routine based on functionality, see Universal C runtime routines by category.

A

abort

abs

_abs64

access

_access

_access_s

acos

acosf

acosh

acoshf

acoshl

acosl

_aligned_free

_aligned_free_dbg

_aligned_malloc

_aligned_malloc_dbg

_aligned_msize

_aligned_msize_dbg

_aligned_offset_malloc

_aligned_offset_malloc_dbg

_aligned_offset_realloc

_aligned_offset_realloc_dbg

_aligned_offset_recalloc

_aligned_offset_recalloc_dbg

_aligned_realloc

_aligned_realloc_dbg

_aligned_recalloc

_aligned_recalloc_dbg

_alloca

_amsg_exit

and

and_eq

asctime

asctime_s

asin

asinf

asinh

asinhf

asinhl

asinl

assert

_assert

_ASSERT

_ASSERT_EXPR

_ASSERTE

atan

atan2

atan2f

atan2l

atanf

atanh

atanhf

atanhl

atanl

atexit

_atodbl

_atodbl_l

atof

_atof_l

B

C

cacosh

cacoshf

cacoshl

cacosl

_callnewh

calloc

_calloc_dbg

carg

cargf

cargl

casin

casinf

casinh

casinhf

casinhl

casinl

catan

catanf

catanh

catanhf

catanhl

catanl

cbrt

cbrtf

cbrtl

ccos

ccosf

ccosh

ccoshf

ccoshl

ccosl

ceil

ceilf

ceill

_cexit

cexp

cexpf

cexpl

cgets

_cgets_s

_cgetws_s

chdir

_chdir

_chdrive

_chgsign

_chgsignf

_chgsignl

chmod

_chmod

chsize

_chsize

_chsize_s

cimag

cimagf

cimagl

_clear87

clearerr

clearerr_s

_clearfp

clock

clog

clog10

clog10f

clog10l

clogf

clogl

close

_close

_commit

compl

_configthreadlocale

conj

conjf

conjl

_control87

__control87_2

_controlfp

_controlfp_s

copysign

_copysign

copysignf

_copysignf

copysignl

_copysignl

cos

cosf

cosh

coshf

coshl

cosl

_countof

cpow

cpowf

cpowl

cprintf

_cprintf

_cprintf_l

_cprintf_p

_cprintf_p_l

_cprintf_s

_cprintf_s_l

cproj

cprojf

cprojl

cputs

_cputs

_cputws

creal

crealf

creall

creat

_creat

_create_locale

_CrtCheckMemory

_CrtDbgBreak

_CrtDbgReport

_CrtDbgReportW

_CrtDoForAllClientObjects

_CrtDumpMemoryLeaks

_CrtGetAllocHook

_CrtGetDumpClient

_CrtGetReportHook

_CrtIsMemoryBlock

_CrtIsValidHeapPointer

_CrtIsValidPointer

_CrtMemCheckpoint

_CrtMemDifference

_CrtMemDumpAllObjectsSince

_CrtMemDumpStatistics

_CrtReportBlockType

_CrtSetAllocHook

_CrtSetBreakAlloc

_CrtSetDbgFlag

_CrtSetDebugFillThreshold

_CrtSetDumpClient

_CrtSetReportFile

_CrtSetReportHook

_CrtSetReportHook2

_CrtSetReportHookW2

_CrtSetReportMode

cscanf

_cscanf

_cscanf_l

_cscanf_s

_cscanf_s_l

csin

csinf

csinh

csinhf

csinhl

csinl

csqrt

csqrtf

csqrtl

ctan

ctanf

ctanh

ctanhf

ctanhl

ctanl

ctime

ctime_s

_ctime32

_ctime32_s

_ctime64

_ctime64_s

_cwait

cwait

_cwprintf

_cwprintf_l

_cwprintf_p

_cwprintf_p_l

_cwprintf_s

_cwprintf_s_l

_cwscanf

_cwscanf_l

_cwscanf_s

_cwscanf_s_l

_CxxThrowException

D

difftime

_difftime32

_difftime64

div

_dup

dup

_dup2

dup2

_dupenv_s

_dupenv_s_dbg

E

_ecvt

ecvt

_ecvt_s

_endthread

_endthreadex

eof

_eof

erf

erfc

erfcf

erfcl

erff

erfl

execl

_execl

execle

_execle

execlp

_execlp

execlpe

_execlpe

execv

_execv

execve

_execve

execvp

_execvp

execvpe

_execvpe

exit

_Exit

_exit

exp

exp2

exp2f

exp2l

_expand

_expand_dbg

expf

expm1

expm1f

expm1l

F

fabs

fabsf

fclose

_fclose_nolock

_fcloseall

fcloseall

_fcvt

fcvt

_fcvt_s

fdim

fdimf

fdiml

fdopen

_fdopen

feclearexcept

fegetenv

fegetexceptflag

fegetround

feholdexcept

feof

feraiseexcept

ferror

fesetenv

fesetexceptflag

fesetround

fetestexcept

feupdateenv

fflush

_fflush_nolock

fgetc

_fgetc_nolock

fgetchar

_fgetchar

fgetpos

fgets

fgetwc

_fgetwc_nolock

_fgetwchar

fgetws

filelength

_filelength

_filelengthi64

fileno

_fileno

_findclose

_findfirst

_findfirst32

_findfirst32i64

_findfirst64

_findfirst64i32

_findfirsti64

_findnext

_findnext32

_findnext32i64

_findnext64

_findnext64i32

_findnexti64

_finite

_finitef

floor

floorf

floorl

flushall

_flushall

fma

fmaf

fmal

fmax

fmaxf

fmaxl

fmin

fminf

fminl

fmod

fmodf

fopen

fopen_s

_fpclass

_fpclassf

fpclassify

_fpieee_flt

_fpreset

fprintf

_fprintf_l

_fprintf_p

_fprintf_p_l

fprintf_s

_fprintf_s_l

fputc

_fputc_nolock

fputchar

_fputchar

fputs

fputwc

_fputwc_nolock

_fputwchar

fputws

fread

_fread_nolock

_fread_nolock_s

fread_s

free

_free_dbg

_free_locale

_freea

freopen

freopen_s

frexp

fscanf

_fscanf_l

fscanf_s

_fscanf_s_l

fseek

_fseek_nolock

_fseeki64

_fseeki64_nolock

fsetpos

_fsopen

_fstat

_fstat32

_fstat32i64

_fstat64

_fstat64i32

_fstati64

ftell

_ftell_nolock

_ftelli64

_ftelli64_nolock

_ftime

_ftime_s

_ftime32

_ftime32_s

_ftime64

_ftime64_s

_fullpath

_fullpath_dbg

_futime

_futime32

_futime64

fwide

fwprintf

_fwprintf_l

_fwprintf_p

_fwprintf_p_l

fwprintf_s

_fwprintf_s_l

fwrite

_fwrite_nolock

fwscanf

_fwscanf_l

fwscanf_s

_fwscanf_s_l

G

gcvt

_gcvt

_gcvt_s

_get_current_locale

_get_daylight

_get_doserrno

_get_dstbias

_get_errno

_get_FMA3_enable

_get_fmode

_get_heap_handle

_get_invalid_parameter_handler

_get_osfhandle

_get_pgmptr

_get_printf_count_output

_get_terminate

_get_thread_local_invalid_parameter_handler

_get_timezone

_get_tzname

_get_unexpected

_get_wpgmptr

getc

_getc_nolock

getch

_getch

_getch_nolock

getchar

_getchar_nolock

getche

_getche

_getche_nolock

getcwd

_getcwd

_getcwd_dbg

_getdcwd

_getdcwd_dbg

_getdcwd_nolock

_getdiskfree

_getdrive

_getdrives

getenv

getenv_s

H

I

_i64toa

_i64toa_s

_i64tow

_i64tow_s

ilogb

ilogbf

ilogbl

imaxabs

imaxdiv

_initterm

_initterm_e

_invalid_parameter

_invalid_parameter_noinfo

_invalid_parameter_noinfo_noreturn

_invoke_watson

isalnum

_isalnum_l

isalpha

_isalpha_l

isascii

__isascii

_isatty

isatty

isblank

_isblank_l

iscntrl

_iscntrl_l

__iscsym

iscsym

_iscsym_l

__iscsymf

iscsymf

_iscsymf_l

_isctype

_isctype_l

isdigit

_isdigit_l

isfinite

isgraph

_isgraph_l

isgreater

isgreaterequal

isinf

isleadbyte

_isleadbyte_l

isless

islessequal

islessgreater

islower

_islower_l

_ismbbalnum

_ismbbalnum_l

_ismbbalpha

_ismbbalpha_l

_ismbbblank

_ismbbblank_l

_ismbbgraph

_ismbbgraph_l

_ismbbkalnum

_ismbbkalnum_l

_ismbbkana

_ismbbkana_l

_ismbbkprint

_ismbbkprint_l

_ismbbkpunct

_ismbbkpunct_l

_ismbblead

_ismbblead_l

_ismbbprint

_ismbbprint_l

_ismbbpunct

_ismbbpunct_l

_ismbbtrail

_ismbbtrail_l

_ismbcalnum

_ismbcalnum_l

_ismbcalpha

_ismbcalpha_l

_ismbcblank

_ismbcblank_l

_ismbcdigit

_ismbcdigit_l

_ismbcgraph

_ismbcgraph_l

_ismbchira

_ismbchira_l

_ismbckata

_ismbckata_l

_ismbcl0

_ismbcl0_l

_ismbcl1

_ismbcl1_l

_ismbcl2

_ismbcl2_l

_ismbclegal

_ismbclegal_l

_ismbclower

_ismbclower_l

_ismbcprint

_ismbcprint_l

_ismbcpunct

_ismbcpunct_l

_ismbcspace

_ismbcspace_l

_ismbcsymbol

_ismbcsymbol_l

_ismbcupper

_ismbcupper_l

_ismbslead

_ismbslead_l

_ismbstrail

_ismbstrail_l

isnan

_isnan

_isnanf

isnormal

isprint

_isprint_l

ispunct

_ispunct_l

isspace

_isspace_l

isunordered

isupper

_isupper_l

iswalnum

_iswalnum_l

iswalpha

_iswalpha_l

iswascii

iswblank

_iswblank_l

iswcntrl

_iswcntrl_l

__iswcsym

_iswcsym_l

__iswcsymf

_iswcsymf_l

iswctype

_iswctype_l

iswdigit

_iswdigit_l

iswgraph

_iswgraph_l

iswlower

_iswlower_l

iswprint

_iswprint_l

iswpunct

_iswpunct_l

iswspace

_iswspace_l

iswupper

_iswupper_l

iswxdigit

_iswxdigit_l

isxdigit

_isxdigit_l

itoa

_itoa

_itoa_s

_itow

_itow_s

J

_j0

j0

_j1

j1

_jn

jn

K

_kbhit

kbhit

L

labs

ldexp

ldiv

_lfind

lfind

_lfind_s

lgamma

lgammaf

lgammal

llabs

lldiv

llrint

llrintf

llrintl

llround

llroundf

llroundl

localeconv

localtime

localtime_s

_localtime32

_localtime32_s

_localtime64

_localtime64_s

_lock_file

locking

_locking

log

log10

log10f

log1p

log1pf

log1pl

log2

log2f

log2l

logb

_logb

logbf

_logbf

logbl

logf

longjmp

lrint

lrintf

lrintl

_lrotl

_lrotr

lround

lroundf

lroundl

_lsearch

lsearch

_lsearch_s

lseek

_lseek

_lseeki64

ltoa

_ltoa

_ltoa_s

_ltow

_ltow_s

M

_makepath

_makepath_s

malloc

_malloc_dbg

_malloca

_matherr

__max

_mbbtombc

_mbbtombc_l

_mbbtype

_mbbtype_l

_mbccpy

_mbccpy_l

_mbccpy_s

_mbccpy_s_l

_mbcjistojms

_mbcjistojms_l

_mbcjmstojis

_mbcjmstojis_l

_mbclen

_mbclen_l

_mbctohira

_mbctohira_l

_mbctokata

_mbctokata_l

_mbctolower

_mbctolower_l

_mbctombb

_mbctombb_l

_mbctoupper

_mbctoupper_l

mblen

_mblen_l

mbrlen

mbrtoc16

mbrtoc32

mbrtowc

_mbsbtype

_mbsbtype_l

_mbscat

_mbscat_s

_mbscat_s_l

_mbschr

_mbschr_l

_mbscmp

_mbscmp_l

_mbscoll

_mbscoll_l

_mbscpy

_mbscpy_s

_mbscpy_s_l

_mbscspn

_mbscspn_l

_mbsdec

_mbsdec_l

_mbsdup

_mbsicmp

_mbsicmp_l

_mbsicoll

_mbsicoll_l

_mbsinc

_mbsinc_l

mbsinit

_mbslen

_mbslen_l

_mbslwr

_mbslwr_l

_mbslwr_s

_mbslwr_s_l

_mbsnbcat

_mbsnbcat_l

_mbsnbcat_s

_mbsnbcat_s_l

_mbsnbcmp

_mbsnbcmp_l

_mbsnbcnt

_mbsnbcnt_l

_mbsnbcoll

_mbsnbcoll_l

_mbsnbcpy

_mbsnbcpy_l

_mbsnbcpy_s

_mbsnbcpy_s_l

_mbsnbicmp

_mbsnbicmp_l

_mbsnbicoll

_mbsnbicoll_l

_mbsnbset

_mbsnbset_l

_mbsnbset_s

_mbsnbset_s_l

_mbsncat

_mbsncat_l

_mbsncat_s

_mbsncat_s_l

_mbsnccnt

_mbsnccnt_l

_mbsncmp

_mbsncmp_l

_mbsncoll

_mbsncoll_l

_mbsncpy

_mbsncpy_l

_mbsncpy_s

_mbsncpy_s_l

_mbsnextc

_mbsnextc_l

_mbsnicmp

_mbsnicmp_l

_mbsnicoll

_mbsnicoll_l

_mbsninc

_mbsninc_l

_mbsnlen

_mbsnlen_l

_mbsnset

_mbsnset_l

_mbsnset_s

_mbsnset_s_l

_mbspbrk

_mbspbrk_l

_mbsrchr

_mbsrchr_l

_mbsrev

_mbsrev_l

mbsrtowcs

mbsrtowcs_s

_mbsset

_mbsset_l

_mbsset_s

_mbsset_s_l

_mbsspn

_mbsspn_l

_mbsspnp

_mbsspnp_l

_mbsstr

_mbsstr_l

_mbstok

_mbstok_l

_mbstok_s

_mbstok_s_l

mbstowcs

_mbstowcs_l

mbstowcs_s

_mbstowcs_s_l

_mbstrlen

_mbstrlen_l

_mbstrnlen

_mbstrnlen_l

_mbsupr

_mbsupr_l

_mbsupr_s

_mbsupr_s_l

mbtowc

_mbtowc_l

memccpy

_memccpy

memchr

memcmp

memcpy

memcpy_s

memicmp

_memicmp

_memicmp_l

memmove

memmove_s

memset

__min

mkdir

_mkdir

_mkgmtime

_mkgmtime32

_mkgmtime64

mktemp

_mktemp

_mktemp_s

mktime

_mktime32

_mktime64

modf

modff

_msize

_msize_dbg

N

nan

nanf

nanl

nearbyint

nearbyintf

nearbyintl

nextafter

O

P

printf_s

_printf_s_l

_purecall

putc

_putc_nolock

putch

_putch

_putch_nolock

putchar

_putchar_nolock

putenv

_putenv

_putenv_s

puts

putw

_putw

putwc

_putwc_nolock

_putwch

_putwch_nolock

putwchar

_putwchar_nolock

_putws

Q

qsort

qsort_s

_query_new_handler

_query_new_mode

quick_exit

R

raise

rand

rand_s

read

_read

realloc

_realloc_dbg

_recalloc

_recalloc_dbg

remainder

remainderf

remainderl

remove

remquo

remquof

remquol

rename

_resetstkoflw

rewind

rint

rintf

rintl

rmdir

_rmdir

rmtmp

_rmtmp

_rotl

_rotl64

_rotr

_rotr64

round

roundf

roundl

_RPT

_RPTF

_RPTFW

_RPTW

_RTC_GetErrDesc

_RTC_NumErrors

_RTC_SetErrorFunc

_RTC_SetErrorFuncW

_RTC_SetErrorType

S

_scalb

scalbln

scalblnf

scalblnl

scalbn

scalbnf

scalbnl

scanf

_scanf_l

scanf_s

_scanf_s_l

_scprintf

_scprintf_l

_scprintf_p

_scprintf_p_l

_scwprintf

_scwprintf_l

_scwprintf_p

_scwprintf_p_l

_searchenv

_searchenv_s

__security_init_cookie

_seh_filter_dll

_seh_filter_exe

_set_abort_behavior

_set_controlfp

_set_doserrno

_set_errno

_set_error_mode

_set_FMA3_enable

_set_fmode

_set_invalid_parameter_handler

_set_new_handler

_set_new_mode

_set_printf_count_output

_set_purecall_handler

_set_se_translator

_set_SSE2_enable

set_terminate

_set_thread_local_invalid_parameter_handler

set_unexpected

setbuf

setjmp

setlocale

_setmaxstdio

_setmbcp

setmode

_setmode

setvbuf

signal

signbit

sin

sinf

sinh

sinhf

sinhl

sinl

snprintf

_snprintf

_snprintf_l

_snprintf_s

_snprintf_s_l

_snscanf

_snscanf_l

_snscanf_s

_snscanf_s_l

_snwprintf

_snwprintf_l

_snwprintf_s

_snwprintf_s_l

_snwscanf

_snwscanf_l

_snwscanf_s

_snwscanf_s_l

sopen

_sopen

_sopen_s

spawnl

_spawnl

spawnle

_spawnle

spawnlp

_spawnlp

spawnlpe

_spawnlpe

spawnv

_spawnv

spawnve

_spawnve

spawnvp

_spawnvp

spawnvpe

_spawnvpe

_splitpath

_splitpath_s

sprintf

_sprintf_l

_sprintf_p

_sprintf_p_l

sprintf_s

_sprintf_s_l

sqrt

sqrtf

sqrtl

srand

sscanf

_sscanf_l

sscanf_s

_sscanf_s_l

_stat

_stat32

_stat32i64

_stat64

_stat64i32

_stati64

_STATIC_ASSERT

_status87

_statusfp

_statusfp2

strcat

strcat_s

strchr

strcmp

strcmpi

strcoll

_strcoll_l

strcpy

strcpy_s

strcspn

_strdate

_strdate_s

_strdec

_strdup

strdup

_strdup_dbg

strerror

_strerror

strerror_s

_strerror_s

strftime

_strftime_l

_stricmp

stricmp

_stricmp_l

_stricoll

_stricoll_l

_strinc

strlen

_strlwr

strlwr

_strlwr_l

_strlwr_s

_strlwr_s_l

strncat

_strncat_l

strncat_s

_strncat_s_l

strncmp

_strncnt

_strncoll

_strncoll_l

strncpy

_strncpy_l

strncpy_s

_strncpy_s_l

_strnextc

_strnicmp

strnicmp

_strnicmp_l

_strnicoll

_strnicoll_l

_strninc

strnlen

strnlen_s

_strnset

strnset

_strnset_l

_strnset_s

_strnset_s_l

strpbrk

strrchr

_strrev

strrev

_strset

strset

_strset_l

_strset_s

_strset_s_l

strspn

_strspnp

strstr

_strtime

_strtime_s

strtod

_strtod_l

strtof

_strtof_l

_strtoi64

_strtoi64_l

strtoimax

_strtoimax_l

strtok

_strtok_l

strtok_s

_strtok_s_l

strtol

_strtol_l

strtold

_strtold_l

strtoll

_strtoll_l

_strtoui64

_strtoui64_l

strtoul

_strtoul_l

strtoull

_strtoull_l

strtoumax

_strtoumax_l

_strupr

strupr

_strupr_l

_strupr_s

_strupr_s_l

strxfrm

_strxfrm_l

swab

_swab

swprintf

_swprintf_l

__swprintf_l

_swprintf_p

_swprintf_p_l

swprintf_s

_swprintf_s_l

swscanf

_swscanf_l

swscanf_s

_swscanf_s_l

system

T

tan

tanf

tanh

tanhf

tanhl

tanl

tell

_tell

_telli64

tempnam

_tempnam

_tempnam_dbg

terminate

tgamma

tgammaf

tgammal

time

_time32

_time64

timespec_get

_timespec32_get

_timespec64_get

tmpfile

tmpfile_s

tmpnam

tmpnam_s

__toascii

toascii

tolower

_tolower

_tolower_l

toupper

_toupper

_toupper_l

towctrans

towlower

_towlower_l

towupper

_towupper_l

trunc

truncf

truncl

tzset

_tzset

U

_ui64toa

_ui64toa_s

_ui64tow

_ui64tow_s

ultoa

_ultoa

_ultoa_s

_ultow

_ultow_s

umask

_umask

_umask_s

__uncaught_exception

unexpected

ungetc

_ungetc_nolock

ungetch

_ungetch

_ungetch_nolock

ungetwc

_ungetwc_nolock

_ungetwch

_ungetwch_nolock

unlink

_unlink

_unlock_file

_utime

_utime32

_utime64

V

va_arg

va_copy

va_end

va_start

_vcprintf

_vcprintf_l

_vcprintf_p

_vcprintf_p_l

_vcprintf_s

_vcprintf_s_l

_vcwprintf

_vcwprintf_l

_vcwprintf_p

_vcwprintf_p_l

_vcwprintf_s

_vcwprintf_s_l

vfprintf

_vfprintf_l

_vfprintf_p

_vfprintf_p_l

vfprintf_s

_vfprintf_s_l

vfscanf

vfscanf_s

vfwprintf

_vfwprintf_l

_vfwprintf_p

_vfwprintf_p_l

vfwprintf_s

_vfwprintf_s_l

vfwscanf

vfwscanf_s

vprintf

_vprintf_l

_vprintf_p

_vprintf_p_l

vprintf_s

_vprintf_s_l

vscanf

vscanf_s

_vscprintf

_vscprintf_l

_vscprintf_p

_vscprintf_p_l

_vscwprintf

_vscwprintf_l

_vscwprintf_p

_vscwprintf_p_l

vsnprintf

_vsnprintf

_vsnprintf_l

vsnprintf_s

_vsnprintf_s

_vsnprintf_s_l

_vsnwprintf

_vsnwprintf_l

_vsnwprintf_s

_vsnwprintf_s_l

vsprintf

_vsprintf_l

_vsprintf_p

_vsprintf_p_l

vsprintf_s

_vsprintf_s_l

vsscanf

vsscanf_s

vswprintf

_vswprintf_l

__vswprintf_l

_vswprintf_p

_vswprintf_p_l

vswprintf_s

_vswprintf_s_l

vswscanf

vswscanf_s

vwprintf

_vwprintf_l

_vwprintf_p

_vwprintf_p_l

vwprintf_s

_vwprintf_s_l

vwscanf

vwscanf_s

W

_waccess

_waccess_s

_wasctime

_wasctime_s

_wassert

_wchdir

_wchmod

_wcreat

_wcreate_locale

wcrtomb

wcrtomb_s

wcscat

wcscat_s

wcschr

wcscmp

wcscoll

_wcscoll_l

wcscpy

wcscpy_s

wcscspn

_wcsdec

_wcsdup

wcsdup

_wcsdup_dbg

_wcserror

__wcserror

_wcserror_s

__wcserror_s

wcsftime

_wcsftime_l

_wcsicmp

wcsicmp

_wcsicmp_l

_wcsicoll

wcsicoll

_wcsicoll_l

_wcsinc

wcslen

_wcslwr

wcslwr

_wcslwr_l

_wcslwr_s

_wcslwr_s_l

wcsncat

_wcsncat_l

wcsncat_s

_wcsncat_s_l

wcsncmp

_wcsncnt

_wcsncoll

_wcsncoll_l

wcsncpy

_wcsncpy_l

wcsncpy_s

_wcsncpy_s_l

_wcsnextc

_wcsnicmp

wcsnicmp

_wcsnicmp_l

_wcsnicoll

_wcsnicoll_l

_wcsninc

wcsnlen

wcsnlen_s

_wcsnset

wcsnset

_wcsnset_l

_wcsnset_s

_wcsnset_s_l

wcspbrk

wcsrchr

_wcsrev

wcsrev

wcsrtombs

wcsrtombs_s

_wcsset

wcsset

_wcsset_l

_wcsset_s

_wcsset_s_l

wcsspn

_wcsspnp

wcsstr

wcstod

_wcstod_l

wcstof

_wcstof_l

_wcstoi64

_wcstoi64_l

wcstoimax

_wcstoimax_l

wcstok

_wcstok_l

wcstok_s

_wcstok_s_l

wcstol

_wcstol_l

wcstold

_wcstold_l

wcstoll

_wcstoll_l

wcstombs

_wcstombs_l

wcstombs_s

_wcstombs_s_l

_wcstoui64

_wcstoui64_l

wcstoul

_wcstoul_l

wcstoull

_wcstoull_l

wcstoumax

_wcstoumax_l

_wcsupr

wcsupr

_wcsupr_l

_wcsupr_s

_wcsupr_s_l

wcsxfrm

_wcsxfrm_l

_wctime

_wctime_s

_wctime32

_wctime32_s

_wctime64

_wctime64_s

wctob

wctomb

_wctomb_l

wctomb_s

_wctomb_s_l

wctrans

wctype

_wdupenv_s

_wdupenv_s_dbg

_wexecl

_wexecle

_wexeclp

_wexeclpe

_wexecv

_wexecve

_wexecvp

_wexecvpe

_wfdopen

_wfindfirst

_wfindfirst32

_wfindfirst32i64

_wfindfirst64

_wfindfirst64i32

_wfindfirsti64

_wfindnext

_wfindnext32

_wfindnext32i64

_wfindnext64

_wfindnext64i32

_wfindnexti64

_wfopen

_wfopen_s

_wfreopen

_wfreopen_s

_wfsopen

_wfullpath

_wfullpath_dbg

_wgetcwd

_wgetcwd_dbg

_wgetdcwd

_wgetdcwd_dbg

_wgetdcwd_nolock

_wgetenv

_wgetenv_s

_wmakepath

_wmakepath_s

wmemchr

wmemcmp

wmemcpy

wmemcpy_s

wmemmove

wmemmove_s

wmemset

_wmkdir

_wmktemp

_wmktemp_s

_wopen

_wperror

_wpopen

wprintf

_wprintf_l

_wprintf_p

_wprintf_p_l

wprintf_s

_wprintf_s_l

_wputenv

_wputenv_s

_wremove

_wrename

_write

write

_wrmdir

wscanf

_wscanf_l

wscanf_s

_wscanf_s_l

_wsearchenv

_wsearchenv_s

_wsetlocale

_wsopen

_wsopen_s

_wspawnl

_wspawnle

_wspawnlp

_wspawnlpe

_wspawnv

_wspawnve

_wspawnvp

_wspawnvpe

_wsplitpath

_wsplitpath_s

_wstat

_wstat32

_wstat32i64

_wstat64

_wstat64i32

_wstati64

_wstrdate

_wstrdate_s

_wstrtime

_wstrtime_s

_wsystem

_wtempnam

_wtempnam_dbg

_wtmpnam

_wtmpnam_s

_wtof

_wtof_l

_wtoi

_wtoi_l

_wtoi64

_wtoi64_l

_wtol

_wtol_l

_wtoll

_wtoll_l

_wunlink

_wutime

_wutime32

_wutime64

X

xor

xor_eq

Y

_y0

y0

_y1

y1

_yn

yn

See also

C Run-Time Library Reference

abort

10/31/2018 • 3 minutes to read • Edit Online

Aborts the current process and returns an error code.

> **NOTE**
> Do not use this method to shut down a Microsoft Store app or Universal Windows Platform (UWP) app, except in testing or debugging scenarios. Programmatic or UI ways to close a Store app are not permitted according to the Microsoft Store policies. For more information, see UWP app lifecycle.

Syntax

```
void abort( void );
```

Return Value

abort does not return control to the calling process. By default, it checks for an abort signal handler and raises `SIGABRT` if one is set. Then **abort** terminates the current process and returns an exit code to the parent process.

Remarks

Microsoft Specific

By default, when an app is built with the debug runtime library, the **abort** routine displays an error message before `SIGABRT` is raised. For console apps running in console mode, the message is sent to `STDERR`. Windows desktop apps and console apps running in windowed mode display the message in a message box. To suppress the message, use _set_abort_behavior to clear the `_WRITE_ABORT_MSG` flag. The message displayed depends on the version of the runtime environment used. For applications built by using the most recent versions of Visual C++, the message resembles this:

```
R6010 - abort() has been called
```

In previous versions of the C runtime library, this message was displayed:

```
This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.
```

When the program is compiled in debug mode, the message box displays options to **Abort**, **Retry**, or **Ignore**. If the user chooses **Abort**, the program terminates immediately and returns an exit code of 3. If the user chooses **Retry**, a debugger is invoked for just-in-time debugging, if available. If the user chooses **Ignore**, **abort** continues normal processing.

In both retail and debug builds, **abort** then checks whether an abort signal handler is set. If a non-default signal handler is set, **abort** calls `raise(SIGABRT)`. Use the signal function to associate an abort signal handler function with the `SIGABRT` signal. You can perform custom actions—for example, clean up resources or log information—and terminate the app with your own error code in the handler function. If

no custom signal handler is defined, **abort** does not raise the `SIGABRT` signal.

By default, in non-debug builds of desktop or console apps, **abort** then invokes the Windows Error Reporting Service mechanism (formerly known as Dr. Watson) to report failures to Microsoft. This behavior can be enabled or disabled by calling `_set_abort_behavior` and setting or masking the `_CALL_REPORTFAULT` flag. When the flag is set, Windows displays a message box that has text something like "A problem caused the program to stop working correctly." The user can choose to invoke a debugger with a **Debug** button, or choose the **Close program** button to terminate the app with an error code that's defined by the operating system.

If the Windows error reporting handler is not invoked, then **abort** calls _exit to terminate the process with exit code 3 and returns control to the parent process or the operating system. `_exit` does not flush stream buffers or do `atexit` / `_onexit` processing.

For more information about CRT debugging, see CRT Debugging Techniques.

End Microsoft Specific

Requirements

ROUTINE	REQUIRED HEADER
abort	<process.h> or <stdlib.h>

Example

The following program tries to open a file and aborts if the attempt fails.

```
// crt_abort.c
// compile with: /TC
// This program demonstrates the use of
// the abort function by attempting to open a file
// and aborts if the attempt fails.

#include  <stdio.h>
#include  <stdlib.h>

int main( void )
{
    FILE    *stream = NULL;
    errno_t err = 0;

    err = fopen_s(&stream, "NOSUCHF.ILE", "r" );
    if ((err != 0) || (stream == NULL))
    {
        perror( "File could not be opened" );
        abort();
    }
    else
    {
        fclose( stream );
    }
}
```

```
File could not be opened: No such file or directory
```

See also

abs, labs, llabs, _abs64

10/31/2018 • 2 minutes to read • Edit Online

Calculates the absolute value of the argument.

Syntax

```
int abs( int n );
long labs( long n );
long long llabs( long long n );
__int64 _abs64( __int64 n );
```

```
long abs( long n );   // C++ only
long long abs( long long n );   // C++ only
double abs( double n );   // C++ only
long double abs( long double n );   // C++ only
float abs( float n );   // C++ only
```

Parameters

n
Numeric value.

Return Value

The **abs**, **labs**, **llabs** and **_abs64** functions return the absolute value of the parameter *n*. There is no error return.

Remarks

Because C++ allows overloading, you can call overloads of **abs** that take and return **long**, **long long**, **float**, **double**, and **long double** values. These overloads are defined in the <cmath> header. In a C program, **abs** always takes and returns an **int**.

Microsoft Specific: Because the range of negative integers that can be represented by using any integral type is larger than the range of positive integers that can be represented by using that type, it's possible to supply an argument to these functions that can't be converted. If the absolute value of the argument cannot be represented by the return type, the **abs** functions return the argument value unchanged. Specifically, `abs(INT_MIN)` returns `INT_MIN`, `labs(LONG_MIN)` returns `LONG_MIN`, `llabs(LLONG_MIN)` returns `LLONG_MIN`, and `_abs64(_I64_MIN)` returns `_I64_MIN`. This means that the **abs** functions cannot be used to guarantee a positive value.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
abs, **labs**, **llabs**	<math.h> or <stdlib.h>	<cmath>, <cstdlib>, <stdlib.h> or <math.h>
_abs64	<stdlib.h>	<cstdlib> or <stdlib.h>

To use the overloaded versions of **abs** in C++, you must include the <cmath> header.

Example

This program computes and displays the absolute values of several numbers.

```
// crt_abs.c
// Build: cl /W3 /TC crt_abs.c
// This program demonstrates the use of the abs function
// by computing and displaying the absolute values of
// several numbers.

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <limits.h>

int main( void )
{
    int ix = -4;
    long lx = -41567L;
    long long llx = -9876543210LL;
    __int64 wx = -1;

    // absolute 32 bit integer value
    printf_s("The absolute value of %d is %d\n", ix, abs(ix));

    // absolute long integer value
    printf_s("The absolute value of %ld is %ld\n", lx, labs(lx));

    // absolute long long integer value
    printf_s("The absolute value of %lld is %lld\n", llx, llabs(llx));

    // absolute 64 bit integer value
    printf_s("The absolute value of 0x%.16I64x is 0x%.16I64x\n", wx,
        _abs64(wx));

    // Integer error cases:
    printf_s("Microsoft implementation-specific results:\n");
    printf_s(" abs(INT_MIN) returns %d\n", abs(INT_MIN));
    printf_s(" labs(LONG_MIN) returns %ld\n", labs(LONG_MIN));
    printf_s(" llabs(LLONG_MIN) returns %lld\n", llabs(LLONG_MIN));
    printf_s(" _abs64(_I64_MIN) returns 0x%.16I64x\n", _abs64(_I64_MIN));
}
```

```
The absolute value of -4 is 4
The absolute value of -41567 is 41567
The absolute value of -9876543210 is 9876543210
The absolute value of 0xffffffffffffffff is 0x0000000000000001
Microsoft implementation-specific results:
abs(INT_MIN) returns -2147483648
labs(LONG_MIN) returns -2147483648
llabs(LLONG_MIN) returns -9223372036854775808
_abs64(_I64_MIN) returns 0x8000000000000000
```

See also

Data Conversion
Floating-Point Support
_cabs
fabs, fabsf, fabsl

imaxabs

access (CRT)

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _access or security-enhanced _access_s instead.

_access, _waccess

10/31/2018 • 2 minutes to read • Edit Online

Determines if a file is read-only or not. More secure versions are available; see _access_s, _waccess_s.

Syntax

```
int _access(
   const char *path,
   int mode
);
int _waccess(
   const wchar_t *path,
   int mode
);
```

Parameters

path
File or directory path.

mode
Read/write attribute.

Return Value

Each function returns 0 if the file has the given mode. The function returns -1 if the named file does not exist or does not have the given mode; in this case, `errno` is set as shown in the following table.

`EACCES`	Access denied: the file's permission setting does not allow specified access.
`ENOENT`	File name or path not found.
`EINVAL`	Invalid parameter.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

When used with files, the **_access** function determines whether the specified file or directory exists and has the attributes specified by the value of *mode*. When used with directories, **_access** determines only whether the specified directory exists; in Windows 2000 and later operating systems, all directories have read and write access.

***MODE* VALUE**	**CHECKS FILE FOR**
00	Existence only
02	Write-only

MODE VALUE	CHECKS FILE FOR
04	Read-only
06	Read and write

This function only checks whether the file and directory are read-only or not, it does not check the filesystem security settings. For that you need an access token. For more information on filesystem security, see Access Tokens. An ATL class exists to provide this functionality; see CAccessToken Class.

_waccess is a wide-character version of **_access**; the *path* argument to **_waccess** is a wide-character string. **_waccess** and **_access** behave identically otherwise.

This function validates its parameters. If *path* is NULL or *mode* does not specify a valid mode, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function sets `errno` to `EINVAL` and returns -1.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_taccess`	**_access**	**_access**	**_waccess**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_access	<io.h>	<errno.h>
_waccess	<wchar.h> or <io.h>	<errno.h>

Example

The following example uses **_access** to check the file named crt_ACCESS.C to see whether it exists and whether writing is allowed.

```
// crt_access.c
// compile with: /W1
// This example uses _access to check the file named
// crt_ACCESS.C to see if it exists and if writing is allowed.

#include  <io.h>
#include  <stdio.h>
#include  <stdlib.h>

int main( void )
{
    // Check for existence.
    if( (_access( "crt_ACCESS.C", 0 )) != -1 )
    {
        printf_s( "File crt_ACCESS.C exists.\n" );

        // Check for write permission.
        // Assume file is read-only.
        if( (_access( "crt_ACCESS.C", 2 )) == -1 )
            printf_s( "File crt_ACCESS.C does not have write permission.\n" );
    }
}
```

```
File crt_ACCESS.C exists.
File crt_ACCESS.C does not have write permission.
```

See also

File Handling
_chmod, _wchmod
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_open, _wopen
_stat, _wstat Functions

_access_s, _waccess_s

10/31/2018 • 2 minutes to read • Edit Online

Determines file read/write permissions. This is a version of _access, _waccess with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _access_s(
   const char *path,
   int mode
);
errno_t _waccess_s(
   const wchar_t *path,
   int mode
);
```

Parameters

path
File or directory path.

mode
Permission setting.

Return Value

Each function returns 0 if the file has the given mode. The function returns an error code if the named file does not exist or is not accessible in the given mode. In this case, the function returns an error code from the set as follows and also sets `errno` to the same value.

ERRNO VALUE	CONDITION
`EACCES`	Access denied. The file's permission setting does not allow specified access.
`ENOENT`	File name or path not found.
`EINVAL`	Invalid parameter.

For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

When used with files, the **_access_s** function determines whether the specified file exists and can be accessed as specified by the value of *mode*. When used with directories, **_access_s** determines only whether the specified directory exists. In Windows 2000 and later operating systems, all directories have read and write access.

MODE VALUE	CHECKS FILE FOR
00	Existence only.

MODE VALUE	CHECKS FILE FOR
02	Write permission.
04	Read permission.
06	Read and write permission.

Permission to read or write the file is not enough to ensure the ability to open a file. For example, if a file is locked by another process, it might not be accessible even though **_access_s** returns 0.

_waccess_s is a wide-character version of **_access_s**, where the *path* argument to **_waccess_s** is a wide-character string. Otherwise, **_waccess_s** and **_access_s** behave identically.

These functions validate their parameters. If *path* is NULL or *mode* does not specify a valid mode, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set `errno` to `EINVAL` and return `EINVAL`.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_taccess_s`	**_access_s**	**_access_s**	**_waccess_s**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_access_s	<io.h>	<errno.h>
_waccess_s	<wchar.h> or <io.h>	<errno.h>

Example

This example uses **_access_s** to check the file named crt_access_s.c to see whether it exists and whether writing is allowed.

```
// crt_access_s.c

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    errno_t err = 0;

    // Check for existence.
    if ((err = _access_s( "crt_access_s.c", 0 )) == 0 )
    {
        printf_s( "File crt_access_s.c exists.\n" );

        // Check for write permission.
        if ((err = _access_s( "crt_access_s.c", 2 )) == 0 )
        {
            printf_s( "File crt_access_s.c does have "
                      "write permission.\n" );
        }
        else
        {
            printf_s( "File crt_access_s.c does not have "
                      "write permission.\n" );
        }
    }
    else
    {
        printf_s( "File crt_access_s.c does not exist.\n" );
    }
}
```

```
File crt_access_s.c exists.
File crt_access_s.c does not have write permission.
```

See also

File Handling
_access, _waccess
_chmod, _wchmod
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_open, _wopen
_stat, _wstat Functions

acos, acosf, acosl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the arccosine.

Syntax

```
double acos( double x );
float acosf( float x );
long double acosl( long double x );
```

```
float acos( float x );   // C++ only
long double acos( long double x );   // C++ only
```

Parameters

x
Value between -1 and 1, for which to calculate the arccosine (the inverse cosine).

Return Value

The **acos** function returns the arccosine of *x* in the range 0 to π radians.

By default, if *x* is less than -1 or greater than 1, **acos** returns an indefinite.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± ∞	INVALID	_DOMAIN
± QNAN,IND	none	_DOMAIN
|x|>1	INVALID	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **acos** that take and return **float** and **long double** types. In a C program, **acos** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
acos, **acosf**, **acosl**	<math.h>	<errno.h>

Example

This program prompts for a value in the range -1 to 1. Input values outside this range produce `_DOMAIN` error messages. If a valid value is entered, the program prints the arcsine and the arccosine of that value.

```
// crt_asincos.c
// arguments: 0

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( int ac, char* av[] )
{
    double  x,
            y;
    errno_t err;

    // argument checking
    if (ac != 2)
    {
        fprintf_s( stderr, "Usage: %s <number between -1 and 1>\n",
                   av[0]);
        return 1;
    }

    // Convert argument into a double value
    if ((err = sscanf_s( av[1], "%lf", &x )) != 1)
    {
        fprintf_s( stderr, "Error converting argument into ",
                   "double value.\n");
        return 1;
    }

    // Arcsine of X
    y = asin( x );
    printf_s( "Arcsine of %f = %f\n", x, y );

    // Arccosine of X
    y = acos( x );
    printf_s( "Arccosine of %f = %f\n", x, y );
}
```

```
Arcsine of 0.000000 = 0.000000
Arccosine of 0.000000 = 1.570796
```

See also

Floating-Point Support
asin, asinf, asinl
atan, atanf, atanl, atan2, atan2f, atan2l
cos, cosf, cosl
_matherr
sin, sinf, sinl
tan, tanf, tanl

acosh, acoshf, acoshl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the inverse hyperbolic cosine.

Syntax

```
double acosh( double x );
float acoshf( float x );
long double acoshl( long double x );
```

```
float acosh( float x );  // C++ only
long double acosh( long double x );  // C++ only
```

Parameters

x
Floating-point value.

Return Value

The **acosh** functions return the inverse hyberbolic cosine (arc hyperbolic cosine) of *x*. These functions are valid over the domain $x \geq 1$. If *x* is less than 1, `errno` is set to `EDOM` and the result is a quiet NaN. If *x* is a quiet NaN, indefinite, or infinity, the same value is returned.

INPUT	SEH EXCEPTION	`_MATHERR` EXCEPTION
± QNAN, IND, INF	none	none
$x < 1$	none	none

Remarks

When you use C++, you can call overloads of **acosh** that take and return **float** or **long double** values. In a C program, **acosh** always takes and returns **double**.

Requirements

FUNCTION	C HEADER	C++ HEADER
acosh, **acoshf**, **acoshl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

Example

```
// crt_acosh.c
// Compile by using: cl /W4 crt_acosh.c
// This program displays the hyperbolic cosine of pi / 4
// and the arc hyperbolic cosine of the result.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = cosh( pi / 4 );
   y = acosh( x );
   printf( "cosh( %f ) = %f\n", pi/4, x );
   printf( "acosh( %f ) = %f\n", x, y );
}
```

```
cosh( 0.785398 ) = 1.324609
acosh( 1.324609 ) = 0.785398
```

See also

Floating-Point Support
asinh, asinhf, asinhl
atanh, atanhf, atanhl
cosh, coshf, coshl
sinh, sinhf, sinhl
tanh, tanhf, tanhl

_aligned_free

10/31/2018 • 2 minutes to read • Edit Online

Frees a block of memory that was allocated with _aligned_malloc or _aligned_offset_malloc.

Syntax

```
void _aligned_free (
   void *memblock
);
```

Parameters

memblock
A pointer to the memory block that was returned to the `_aligned_malloc` or `_aligned_offset_malloc` function.

Remarks

_aligned_free is marked `__declspec(noalias)`, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

This function does not validate its parameter, unlike the other _aligned CRT functions. If *memblock* is a NULL pointer, this function simply performs no actions. It does not change `errno` and it does not invoke the invalid parameter handler. If an error occurs in the function due to not using _aligned functions previously to allocate the block of memory or a misalignment of memory occurs due to some unforeseen calamity, the function generates a debug report from the _RPT, _RPTF, _RPTW, _RPTFW Macros.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_free	<malloc.h>

Example

For more information, see _aligned_malloc.

See also

Data Alignment

_aligned_free_dbg

10/31/2018 • 2 minutes to read • Edit Online

Frees a block of memory that was allocated with _aligned_malloc or _aligned_offset_malloc (debug only).

Syntax

```
void _aligned_free_dbg(
   void *memblock
);
```

Parameters

memblock
A pointer to the memory block that was returned to the _aligned_malloc or _aligned_offset_malloc function.

Remarks

The **_aligned_free_dbg** function is a debug version of the _aligned_free function. When _DEBUG is not defined, each call to **_aligned_free_dbg** is reduced to a call to `_aligned_free`. Both `_aligned_free` and **_aligned_free_dbg** free a memory block in the base heap, but **_aligned_free_dbg** accommodates a debugging feature: the ability to keep freed blocks in the heap's linked list to simulate low memory conditions.

_aligned_free_dbg performs a validity check on all specified files and block locations before performing the free operation. The application is not expected to provide this information. When a memory block is freed, the debug heap manager automatically checks the integrity of the buffers on either side of the user portion and issues an error report if overwriting has occurred. If the _CRTDBG_DELAY_FREE_MEM_DF bit field of the _crtDbgFlag flag is set, the freed block is filled with the value 0xDD, assigned the _FREE_BLOCK block type, and kept in the heap's linked list of memory blocks.

If an error occurs in freeing the memory, `errno` is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_free_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

See also

Debug Routines

_aligned_malloc

10/31/2018 • 2 minutes to read • Edit Online

Allocates memory on a specified alignment boundary.

Syntax

```
void * _aligned_malloc(
    size_t size,
    size_t alignment
);
```

Parameters

size
Size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

Return Value

A pointer to the memory block that was allocated or NULL if the operation failed. The pointer is a multiple of *alignment*.

Remarks

_aligned_malloc is based on malloc.

_aligned_malloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

This function sets `errno` to `ENOMEM` if the memory allocation failed or if the requested size was greater than `_HEAP_MAXREQ`. For more information about `errno`, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_malloc** validates its parameters. If *alignment* is not a power of 2 or *size* is zero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns NULL and sets `errno` to `EINVAL`.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_malloc	<malloc.h>

Example

```
// crt_aligned_malloc.c

#include <malloc.h>
#include <stdio.h>
```

```
int main() {
    void    *ptr;
    size_t  alignment,
            off_set;

    // Note alignment should be 2^N where N is any positive int.
    alignment = 16;
    off_set = 5;

    // Using _aligned_malloc
    ptr = _aligned_malloc(100, alignment);
    if (ptr == NULL)
    {
        printf_s( "Error allocation aligned memory.");
        return -1;
    }
    if (((unsigned long long)ptr % alignment ) == 0)
        printf_s( "This pointer, %p, is aligned on %zu\n",
                  ptr, alignment);
    else
        printf_s( "This pointer, %p, is not aligned on %zu\n",
                  ptr, alignment);

    // Using _aligned_realloc
    ptr = _aligned_realloc(ptr, 200, alignment);
    if ( ((unsigned long long)ptr % alignment ) == 0)
        printf_s( "This pointer, %p, is aligned on %zu\n",
                  ptr, alignment);
    else
        printf_s( "This pointer, %p, is not aligned on %zu\n",
                  ptr, alignment);
    _aligned_free(ptr);

    // Using _aligned_offset_malloc
    ptr = _aligned_offset_malloc(200, alignment, off_set);
    if (ptr == NULL)
    {
        printf_s( "Error allocation aligned offset memory.");
        return -1;
    }
    if ( ( (((unsigned long long)ptr) + off_set) % alignment ) == 0)
        printf_s( "This pointer, %p, is offset by %zu on alignment of %zu\n",
                  ptr, off_set, alignment);
    else
        printf_s( "This pointer, %p, does not satisfy offset %zu "
                  "and alignment %zu\n",ptr, off_set, alignment);

    // Using _aligned_offset_realloc
    ptr = _aligned_offset_realloc(ptr, 200, alignment, off_set);
    if (ptr == NULL)
    {
        printf_s( "Error reallocation aligned offset memory.");
        return -1;
    }
    if ( ( (((unsigned long long)ptr) + off_set) % alignment ) == 0)
        printf_s( "This pointer, %p, is offset by %zu on alignment of %zu\n",
                  ptr, off_set, alignment);
    else
        printf_s( "This pointer, %p, does not satisfy offset %zu and "
                  "alignment %zu\n", ptr, off_set, alignment);

    // Note that _aligned_free works for both _aligned_malloc
    // and _aligned_offset_malloc. Using free is illegal.
    _aligned_free(ptr);
}
```

```
This pointer, 3280880, is aligned on 16
This pointer, 3280880, is aligned on 16
This pointer, 3280891, is offset by 5 on alignment of 16
This pointer, 3280891, is offset by 5 on alignment of 16
```

See also

Data Alignment

_aligned_malloc_dbg

12/5/2018 • 2 minutes to read • Edit Online

Allocates memory on a specified alignment boundary with additional space for a debugging header and overwrite buffers (debug version only).

Syntax

```
void * _aligned_malloc_dbg(
    size_t size,
    size_t alignment,
   const char *filename,
   int linenumber
);
```

Parameters

size
Size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

filename
Pointer to the name of the source file that requested the allocation operation or NULL.

linenumber
Line number in the source file where the allocation operation was requested or NULL.

Return Value

A pointer to the memory block that was allocated or NULL if the operation failed.

Remarks

_aligned_malloc_dbg is a debug version of the _aligned_malloc function. When _DEBUG is not defined, each call to **_aligned_malloc_dbg** is reduced to a call to `_aligned_malloc`. Both `_aligned_malloc` and **_aligned_malloc_dbg** allocate a block of memory in the base heap, but **_aligned_malloc_dbg** offers several debugging features: buffers on either side of the user portion of the block to test for leaks, and *filename/linenumber* information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

_aligned_malloc_dbg allocates the memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

_aligned_malloc_dbg sets `errno` to `ENOMEM` if a memory allocation fails or if the amount of memory needed (including the overhead mentioned previously) exceeds `_HEAP_MAXREQ`. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_malloc_dbg** validates its parameters. If *alignment* is not a power of 2 or *size* is zero, this function invokes the invalid parameter handler, as described in

Parameter Validation. If execution is allowed to continue, this function returns NULL and sets `errno` to `EINVAL`.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_malloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

_aligned_msize

10/31/2018 • 2 minutes to read • Edit Online

Returns the size of a memory block allocated in the heap.

Syntax

```
size_t _msize(
   void *memblock,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock
Pointer to the memory block.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

Return Value

Returns the size (in bytes) as an unsigned integer.

Remarks

The **_aligned_msize** function returns the size, in bytes, of the memory block allocated by a call to _aligned_malloc or _aligned_realloc. The *alignment* and *offset* values must be the same as the values passed to the function that allocated the block.

When the application is linked with a debug version of the C run-time libraries, **_aligned_msize** resolves to _aligned_msize_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

This function validates its parameter. If *memblock* is a null pointer or *alignment* is not a power of 2, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

Requirements

ROUTINE	REQUIRED HEADER
_msize	<malloc.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Memory Allocation

_aligned_msize_dbg

10/31/2018 • 2 minutes to read • Edit Online

Returns the size of a memory block allocated in the heap (debug version only).

Syntax

```
size_t _aligned_msize_dbg(
   void *memblock,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock
Pointer to the memory block.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

Return Value

Returns the size (in bytes) as an unsigned integer.

Remarks

The *alignment* and *offset* values must be the same as the values passed to the function that allocated the block.

_aligned_msize_dbg is a debug version of the _aligned_msize function. When _DEBUG is not defined, each call to **_aligned_msize_dbg** is reduced to a call to **_aligned_msize**. Both **_aligned_msize** and **_aligned_msize_dbg** calculate the size of a memory block in the base heap, but **_aligned_msize_dbg** adds a debugging feature: It includes the buffers on either side of the user portion of the memory block in the returned size.

This function validates its parameter. If *memblock* is a null pointer or *alignment* is not a power of 2, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_msize_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Memory Allocation

_aligned_offset_malloc

10/31/2018 • 2 minutes to read • Edit Online

Allocates memory on a specified alignment boundary.

Syntax

```
void * _aligned_offset_malloc(
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

size
The size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

Return Value

A pointer to the memory block that was allocated or **NULL** if the operation failed.

Remarks

_aligned_offset_malloc is useful in situations where alignment is needed on a nested element; for example, if alignment was needed on a nested class.

_aligned_offset_malloc is based on **malloc**; for more information, see malloc.

_aligned_offset_malloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_malloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_malloc	<malloc.h>

Example

For more information, see _aligned_malloc.

See also

Data Alignment

_aligned_offset_malloc_dbg

12/5/2018 • 2 minutes to read • Edit Online

Allocates memory on a specified alignment boundary (debug version only).

Syntax

```
void * _aligned_offset_malloc_dbg(
   size_t size,
   size_t alignment,
   size_t offset,
   const char *filename,
   int linenumber
);
```

Parameters

size
The size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

filename
Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber
Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

A pointer to the memory block that was allocated or **NULL** if the operation failed.

Remarks

_aligned_offset_malloc_dbg is a debug version of the _aligned_offset_malloc function. When _DEBUG is not defined, each call to **_aligned_offset_malloc_dbg** is reduced to a call to **_aligned_offset_malloc**. Both **_aligned_offset_malloc** and **_aligned_offset_malloc_dbg** allocate a block of memory in the base heap, but **_aligned_offset_malloc_dbg** offers several debugging features: buffers on either side of the user portion of the block to test for leaks, and *filename*/*linenumber* information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

_aligned_offset_malloc_dbg allocates the memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

_aligned_offset_malloc_dbg is useful in situations where alignment is needed on a nested element; for example, if alignment was needed on a nested class.

_aligned_offset_malloc_dbg is based on **malloc**; for more information, see malloc.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_malloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

For information about the allocation block types and how they are used, see Types of blocks on the debug heap.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_malloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

_aligned_offset_realloc

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.

Syntax

```
void * _aligned_offset_realloc(
   void *memblock,
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock
The current memory block pointer.

size
The size of the memory allocation.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

Return Value

_aligned_offset_realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

_aligned_offset_realloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

Remarks

Like _aligned_offset_malloc, **_aligned_offset_realloc** allows a structure to be aligned at an offset within the structure.

_aligned_offset_realloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc. If *memblock* is **NULL**, the function calls **_aligned_offset_malloc** internally.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_realloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_realloc	<malloc.h>

Example

For more information, see _aligned_malloc.

See also

Data Alignment

_aligned_offset_realloc_dbg

12/5/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc (debug version only).

Syntax

```
void * _aligned_offset_realloc_dbg(
   void *memblock,
   size_t size,
   size_t alignment,
   size_t offset,
   const char *filename,
   int linenumber
);
```

Parameters

memblock
The current memory block pointer.

size
The size of the memory allocation.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

filename
Pointer to the name of the source file that requested the **aligned_offset_realloc** operation or **NULL**.

linenumber
Line number in the source file where the **aligned_offset_realloc** operation was requested or **NULL**.

Return Value

_aligned_offset_realloc_dbg returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

Remarks

_aligned_offset_realloc_dbg is a debug version of the _aligned_offset_realloc function. When _DEBUG is not defined, each call to **_aligned_offset_realloc_dbg** is reduced to a call to **_aligned_offset_realloc**. Both **_aligned_offset_realloc** and **_aligned_offset_realloc_dbg** reallocate a memory block in the base heap, but **_aligned_offset_realloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, and *filename*/*linenumber* information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for

aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

Like _aligned_offset_malloc, **_aligned_offset_realloc_dbg** allows a structure to be aligned at an offset within the structure.

_realloc_dbg reallocates the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. If the memory block is moved, the contents of the original block are overwritten.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_realloc_dbg** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to *size* and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_realloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

_aligned_offset_recalloc

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.

Syntax

```
void * _aligned_offset_recalloc(
   void *memblock,
   size_t num,
   size_t size,
   size_t alignment,
   size_t offset
);
```

Parameters

memblock
The current memory block pointer.

number
Number of elements.

size
Length in bytes of each element.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

Return Value

_aligned_offset_recalloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

_aligned_offset_recalloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables and that the pointer returned is not aliased. For more information, see noalias and restrict.

Remarks

Like _aligned_offset_malloc, **_aligned_offset_recalloc** allows a structure to be aligned at an offset within the structure.

_aligned_offset_recalloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc. If *memblock* is **NULL**, the function calls **_aligned_offset_malloc** internally.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size (*number* * *size*) was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_recalloc** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to the requested size and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_recalloc	<malloc.h>

See also

Data Alignment
_recalloc
_aligned_recalloc

_aligned_offset_recalloc_dbg

12/5/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0 (debug version only).

Syntax

```
void * _aligned_offset_recalloc_dbg(
   void *memblock,
   size_t num,
   size_t size,
   size_t alignment,
   size_t offset,
   const char *filename,
   int linenumber
);
```

Parameters

memblock
The current memory block pointer.

number
Number of elements.

size
Length in bytes of each element.

alignment
The alignment value, which must be an integer power of 2.

offset
The offset into the memory allocation to force the alignment.

filename
Pointer to the name of the source file that requested the realloc operation or **NULL**.

linenumber
Line number in the source file where the realloc operation was requested or **NULL**.

Return Value

_aligned_offset_recalloc_dbg returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

Remarks

_aligned_offset_realloc_dbg is a debug version of the _aligned_offset_recalloc function. When _DEBUG is not defined, each call to **_aligned_offset_recalloc_dbg** is reduced to a call to **_aligned_offset_recalloc**. Both

_aligned_offset_recalloc and **_aligned_offset_recalloc_dbg** reallocate a memory block in the base heap, but **_aligned_offset_recalloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, and *filename/linenumber* information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

_aligned_offset_realloc_dbg reallocates the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. If the memory block is moved, the contents of the original block are overwritten.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size (*number* * *size*) was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_offset_recalloc_dbg** validates its parameters. If *alignment* is not a power of 2 or if *offset* is greater than or equal to the requested size and nonzero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_offset_recalloc_dbg	<malloc.h>

See also

Data Alignment

_aligned_realloc

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc.

Syntax

```
void * _aligned_realloc(
   void *memblock,
   size_t size,
   size_t alignment
);
```

Parameters

memblock
The current memory block pointer.

size
The size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

Return Value

_aligned_realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_realloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_realloc** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_realloc	<malloc.h>

Example

For more information, see _aligned_malloc.

See also

Data Alignment

_aligned_realloc_dbg

12/5/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc (debug version only).

Syntax

```
void * _aligned_realloc_dbg(
   void *memblock,
   size_t size,
   size_t alignment,
   const char *filename,
   int linenumber
);
```

Parameters

memblock
The current memory block pointer.

size
The size of the requested memory allocation.

alignment
The alignment value, which must be an integer power of 2.

filename
Pointer to the name of the source file that requested the **realloc** operation or **NULL**.

linenumber
Line number in the source file where the **realloc** operation was requested or **NULL**.

Return Value

_aligned_realloc_dbg returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_realloc_dbg is a debug version of the _aligned_realloc function. When _DEBUG is not defined, each call to **_aligned_realloc_dbg** is reduced to a call to **_aligned_realloc**. Both **_aligned_realloc** and **_aligned_realloc_dbg** reallocate a memory block in the base heap, but **_aligned_realloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, and *filename/linenumber* information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

_aligned_realloc_dbg reallocates the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. If the memory block is moved, the contents of the original block are overwritten.

_aligned_realloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Also, **_aligned_realloc_dbg** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_realloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

_aligned_recalloc

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0.

Syntax

```
void * _aligned_recalloc(
   void *memblock,
   size_t num,
   size_t size,
   size_t alignment
);
```

Parameters

memblock
The current memory block pointer.

number
The number of elements.

size
The size in bytes of each element.

alignment
The alignment value, which must be an integer power of 2.

Return Value

_aligned_recalloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_recalloc is based on **malloc**. For more information about using **_aligned_offset_malloc**, see malloc.

This function sets **errno** to **ENOMEM** if the memory allocation failed or if the requested size was greater than **_HEAP_MAXREQ**. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr. Also, **_aligned_recalloc** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_recalloc	<malloc.h>

See also

Data Alignment
_recalloc
_aligned_offset_recalloc

_aligned_recalloc_dbg

12/5/2018 • 3 minutes to read • Edit Online

Changes the size of a memory block that was allocated with _aligned_malloc or _aligned_offset_malloc and initializes the memory to 0 (debug version only).

Syntax

```
void * _aligned_recalloc_dbg(
   void * memblock,
   size_t num,
   size_t size,
   size_t alignment,
   const char *filename,
   int linenumber
);
```

Parameters

memblock
The current memory block pointer.

number
The number of elements.

size
The size in bytes of each element.

alignment
The alignment value, which must be an integer power of 2.

filename
Pointer to name of the source file that requested allocation operation or **NULL**.

linenumber
Line number in the source file where allocation operation was requested or **NULL**.

Return Value

_aligned_recalloc_dbg returns a void pointer to the reallocated (and possibly moved) memory block. The return value is **NULL** if the size is zero and the buffer argument is not **NULL**, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second case, the original block is unchanged. The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

It is an error to reallocate memory and change the alignment of a block.

Remarks

_aligned_recalloc_dbg is a debug version of the _aligned_recalloc function. When _DEBUG is not defined, each call to **_aligned_recalloc_dbg** is reduced to a call to **_aligned_recalloc**. Both **_aligned_recalloc** and **_aligned_recalloc_dbg** reallocate a memory block in the base heap, but **_aligned_recalloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, and

filename/linenumber information to determine the origin of allocation requests. Tracking specific allocation types with a block type parameter is not a supported debug feature for aligned allocations. Aligned allocations will appear as a _NORMAL_BLOCK block type.

_aligned_recalloc_dbg reallocates the specified memory block with slightly more space than the requested size (*number* * *size*) which might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. The user portion of the block is filled with the value 0xCD and the overwrite buffers are filled with 0xFD.

_aligned_recalloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails; **EINVAL** is returned if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Also, **_aligned_recalloc_dbg** validates its parameters. If *alignment* is not a power of 2, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns **NULL** and sets **errno** to **EINVAL**.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_aligned_recalloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

_alloca

10/31/2018 • 2 minutes to read • Edit Online

Allocates memory on the stack. This function is deprecated because a more secure version is available; see _malloca.

Syntax

```
void *_alloca(
   size_t size
);
```

Parameters

size
Bytes to be allocated from the stack.

Return Value

The **_alloca** routine returns a **void** pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. If *size* is 0, **_alloca** allocates a zero-length item and returns a valid pointer to that item.

A stack overflow exception is generated if the space cannot be allocated. The stack overflow exception is not a C++ exception; it is a structured exception. Instead of using C++ exception handling, you must use Structured Exception Handling (SEH).

Remarks

_alloca allocates *size* bytes from the program stack. The allocated space is automatically freed when the calling function exits (not when the allocation merely passes out of scope). Therefore, do not pass the pointer value returned by **_alloca** as an argument to free.

There are restrictions to explicitly calling **_alloca** in an exception handler (EH). EH routines that run on x86-class processors operate in their own memory frame: They perform their tasks in memory space that is not based on the current location of the stack pointer of the enclosing function. The most common implementations include Windows NT structured exception handling (SEH) and C++ catch clause expressions. Therefore, explicitly calling **_alloca** in any of the following scenarios results in program failure during the return to the calling EH routine:

- Windows NT SEH exception filter expression: `__except ( _alloca() )`
- Windows NT SEH final exception handler: `__finally { _alloca() }`
- C++ EH catch clause expression

However, **_alloca** can be called directly from within an EH routine or from an application-supplied callback that gets invoked by one of the EH scenarios previously listed.

> **IMPORTANT**
> In Windows XP, if **_alloca** is called inside a try/catch block, you must call _resetstkoflw in the catch block.

In addition to the above restrictions, when using the/clr (Common Language Runtime Compilation) option, **_alloca** cannot be used in **__except** blocks. For more information, see /clr Restrictions.

Requirements

ROUTINE	REQUIRED HEADER
_alloca	<malloc.h>

Example

```
// crt_alloca.c
// This program demonstrates the use of
// _alloca and trapping any exceptions
// that may occur.

#include <windows.h>
#include <stdio.h>
#include <malloc.h>

int main()
{
    int     size = 1000;
    int     errcode = 0;
    void    *pData = NULL;

    // Note: Do not use try/catch for _alloca,
    // use __try/__except, since _alloca throws
    // Structured Exceptions, not C++ exceptions.

    __try {
        // An unbounded _alloca can easily result in a
        // stack overflow.
        // Checking for a size < 1024 bytes is recommended.
        if (size > 0 && size < 1024)
        {
            pData = _alloca( size );
            printf_s( "Allocated %d bytes of stack at 0x%p",
                      size, pData);
        }
        else
        {
            printf_s("Tried to allocate too many bytes.\n");
        }
    }

    // If an exception occured with the _alloca function
    __except( GetExceptionCode() == STATUS_STACK_OVERFLOW )
    {
        printf_s("_alloca failed!\n");

        // If the stack overflows, use this function to restore.
        errcode = _resetstkoflw();
        if (errcode)
        {
            printf_s("Could not reset the stack!\n");
            _exit(1);
        }
    };
}
```

```
Allocated 1000 bytes of stack at 0x0012FB50
```

See also

Memory Allocation
calloc
malloc
realloc
_resetstkoflw
_malloca

_amsg_exit

10/31/2018 • 2 minutes to read • Edit Online

Emits a specified runtime error message and then exits your application with error code 255.

Syntax

```
void _amsg_exit ( int rterrnum );
```

Parameters

rterrnum
The identification number of a system-defined runtime error message.

Remarks

This function emits the runtime error message to **stderr** for console applications, or displays the message in a message box for Windows applications. In debug mode, you can choose to invoke the debugger before exiting.

Requirements

ROUTINE	REQUIRED HEADER
_amsg_exit	internal.h

and

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the && operator.

Syntax

```
#define and &&
```

Remarks

The macro yields the operator &&.

Example

```
// iso646_and.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   bool a = true, b = false, result;

   boolalpha(cout);

   result= a && b;
   cout << result << endl;

   result= a and b;
   cout << result << endl;
}
```

```
false
false
```

Requirements

Header: <iso646.h>

and_eq

11/8/2018 • 2 minutes to read • Edit Online

An alternative to the &= operator.

Syntax

```
#define and_eq &=
```

Remarks

The macro yields the operator &=.

Example

```
// iso646_and_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a &= b;
   cout << result << endl;

   result= a and_eq b;
   cout << result << endl;
}
```

```
2
2
```

Requirements

Header: <iso646.h>

asctime, _wasctime

10/31/2018 • 2 minutes to read • Edit Online

Convert a **tm** time structure to a character string. More secure versions of these functions are available; see asctime_s, _wasctime_s.

Syntax

```
char *asctime(
   const struct tm *timeptr
);
wchar_t *_wasctime(
   const struct tm *timeptr
);
```

Parameters

timeptr
Time/date structure.

Return Value

asctime returns a pointer to the character string result; **_wasctime** returns a pointer to the wide-character string result. There is no error return value.

Remarks

More secure versions of these functions are available; see asctime_s, _wasctime_s.

The **asctime** function converts a time stored as a structure to a character string. The *timeptr* value is usually obtained from a call to **gmtime** or **localtime**, which both return a pointer to a **tm** structure, defined in TIME.H.

TIMEPTR MEMBER	VALUE
tm_hour	Hours since midnight (0-23)
tm_isdst	Positive if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative if status of daylight saving time is unknown. The C run-time library assumes the United States' rules for implementing the calculation of Daylight Saving Time (DST).
tm_mday	Day of month (1-31)
tm_min	Minutes after hour (0-59)
tm_mon	Month (0-11; January = 0)
tm_sec	Seconds after minute (0-59)
tm_wday	Day of week (0-6; Sunday = 0)

TIMEPTR MEMBER	VALUE
tm_yday	Day of year (0-365; January 1 = 0)
tm_year	Year (current year minus 1900)

The converted character string is also adjusted according to the local time zone settings. For information about configuring the local time, see the time, _ftime, and localtime functions and the _tzset function for information about defining the time zone environment and global variables.

The string result produced by **asctime** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used. All fields have a constant width. The newline character and the null character occupy the last two positions of the string. **asctime** uses a single, statically allocated buffer to hold the return string. Each call to this function destroys the result of the previous call.

_wasctime is a wide-character version of **asctime**. **_wasctime** and **asctime** behave identically otherwise.

These functions validate their parameters. If *timeptr* is a null pointer, or if it contains out-of-range values, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

Generic-Text Routine Mapping

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tasctime	**asctime**	**asctime**	**_wasctime**

Requirements

ROUTINE	REQUIRED HEADER
asctime	<time.h>
_wasctime	<time.h> or <wchar.h>

Example

This program places the system time in the long integer **aclock**, translates it into the structure **newtime** and then converts it to string form for output, using the **asctime** function.

```
// crt_asctime.c
// compile with: /W3

#include <time.h>
#include <stdio.h>

int main( void )
{
    struct tm   *newTime;
    time_t      szClock;

    // Get time in seconds
    time( &szClock );

    // Convert time to struct tm form
    newTime = localtime( &szClock );

    // Print local time as a string.
    printf_s( "Current date and time: %s", asctime( newTime ) ); // C4996
    // Note: asctime is deprecated; consider using asctime_s instead
}
```

```
Current date and time: Sun Feb 03 11:38:58 2002
```

See also

Time Management
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64
_tzset
asctime_s, _wasctime_s

asctime_s, _wasctime_s

10/31/2018 • 3 minutes to read • Edit Online

Convert a **tm** time structure to a character string. These functions are versions of asctime, _wasctime with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t asctime_s(
   char* buffer,
   size_t numberOfElements,
   const struct tm *tmSource
);
errno_t _wasctime_s(
   wchar_t* buffer,
   size_t numberOfElements
   const struct tm *tmSource
);
template <size_t size>
errno_t asctime_s(
   char (&buffer)[size],
   const struct tm *tmSource
); // C++ only
template <size_t size>
errno_t _wasctime_s(
   wchar_t (&buffer)[size],
   const struct tm *tmSource
); // C++ only
```

Parameters

buffer
A pointer to a buffer to store the character string result. This function assumes a pointer to a valid memory location with a size specified by *numberOfElements*.

numberOfElements
The size of the buffer used to store the result.

tmSource
Time/date structure. This function assumes a pointer to a valid **struct tm** object.

Return Value

Zero if successful. If there is a failure, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the return value is an error code. Error codes are defined in ERRNO.H. For more information, see errno Constants. The actual error codes returned for each error condition are shown in the following table.

Error Conditions

BUFFER	*NUMBEROFELEMENTS*	*TMSOURCE*	RETURN	VALUE IN *BUFFER*
NULL	Any	Any	**EINVAL**	Not modified

BUFFER	*NUMBEROFELEMENTS*	*TMSOURCE*	RETURN	VALUE IN *BUFFER*
Not **NULL** (points to valid memory)	0	Any	**EINVAL**	Not modified
Not **NULL**	0< size < 26	Any	**EINVAL**	Empty string
Not **NULL**	>= 26	**NULL**	**EINVAL**	Empty string
Not **NULL**	>= 26	Invalid time structure or out of range values for components of the time	**EINVAL**	Empty string

> **NOTE**
> Error conditions for **wasctime_s** are similar to **asctime_s** with the exception that the size limit is measured in words.

Remarks

The **asctime** function converts a time stored as a structure to a character string. The *tmSource* value is usually obtained from a call to **gmtime** or **localtime**. Both functions can be used to fill in a **tm** structure, as defined in TIME.H.

TIMEPTR MEMBER	VALUE
tm_hour	Hours since midnight (0-23)
tm_isdst	Positive if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative if status of daylight saving time is unknown. The C run-time library assumes the United States' rules for implementing the calculation of Daylight Saving Time (DST).
tm_mday	Day of month (1-31)
tm_min	Minutes after hour (0-59)
tm_mon	Month (0-11; January = 0)
tm_sec	Seconds after minute (0-59)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_year	Year (current year minus 1900)

The converted character string is also adjusted according to the local time zone settings. See the time, _time32, _time64, _ftime, _ftime32, _ftime64, and localtime_s, _localtime32_s, _localtime64_s functions for information about configuring the local time and the _tzset function for information about defining the time zone environment and global variables.

The string result produced by **asctime_s** contains exactly 26 characters and has the form `Wed Jan 02 02:03:55 1980\n\0`. A 24-hour clock is used. All fields have a constant width. The new line character and the null character occupy the last two positions of the string. The value passed in as the second parameter should be at least this big. If it is less, an error code, **EINVAL**, will be returned.

_wasctime_s is a wide-character version of **asctime_s**. **_wasctime_s** and **asctime_s** behave identically otherwise.

Generic-Text Routine Mapping

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tasctime_s	**asctime_s**	**asctime_s**	**_wasctime_s**

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
asctime_s	<time.h>
_wasctime_s	<time.h> or <wchar.h>

Security

If the buffer pointer is not **NULL** and the pointer does not point to a valid buffer, the function will overwrite whatever is at the location. This can also result in an access violation.

A buffer overrun can occur if the size argument passed in is greater than the actual size of the buffer.

Example

This program places the system time in the long integer **aclock**, translates it into the structure **newtime** and then converts it to string form for output, using the **asctime_s** function.

```
// crt_asctime_s.c
#include <time.h>
#include <stdio.h>

struct tm newtime;
__time32_t aclock;

int main( void )
{
   char buffer[32];
   errno_t errNum;
   _time32( &aclock );   // Get time in seconds.
   _localtime32_s( &newtime, &aclock );   // Convert time to struct tm form.

   // Print local time as a string.

   errNum = asctime_s(buffer, 32, &newtime);
   if (errNum)
   {
       printf("Error code: %d", (int)errNum);
       return 1;
   }
   printf( "Current date and time: %s", buffer );
   return 0;
}
```

```
Current date and time: Wed May 14 15:30:17 2003
```

See also

Time Management
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_ftime, _ftime32, _ftime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64
_tzset

asin, asinf, asinl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the arcsine.

Syntax

```
double asin( double x );
float asinf ( float x );
long double asinl( long double x );
```

```
float asin( float x );  // C++ only
long double asin( long double x );  // C++ only
```

Parameters

x
Value whose arcsine is to be calculated.

Return Value

The **asin** function returns the arcsine (the inverse sine function) of *x* in the range -π/2 to π/2 radians.

By default, if *x* is less than -1 or greater than 1, **asin** returns an indefinite.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± ∞	**INVALID**	**_DOMAIN**
± **QNAN**, **IND**	none	**_DOMAIN**
|x|>1	**INVALID**	**_DOMAIN**

Remarks

Because C++ allows overloading, you can call overloads of **asin** with **float** and **long double** values. In a C program, **asin** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
asin, **asinf**, **asinl**	<math.h>	<cmath> or <math.h>

Example

For more information, see acos, acosf, acosl.

See also

Floating-Point Support
acos, acosf, acosl
atan, atanf, atanl, atan2, atan2f, atan2l
cos, cosf, cosl
_matherr
sin, sinf, sinl
tan, tanf, tanl

asinh, asinhf, asinhl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the inverse hyperbolic sine.

Syntax

```
double asinh( double x );
float asinhf( float x );
long double asinhl( long double x );
```

```
float asinh( float x );  // C++ only
long double asinh( long double x );  // C++ only
```

Parameters

x
Floating-point value.

Return Value

The **asinh** functions return the inverse hyberbolic sine (arc hyperbolic sine) of *x*. This function is valid over the floating-point domain. If *x* is a quiet NaN, indefinite, or infinity, the same value is returned.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± QNAN, IND, INF	none	none

Remarks

When you use C++, you can call overloads of **asinh** that take and return **float** or **long double** values. In a C program, **asinh** always takes and returns **double**.

Requirements

FUNCTION	REQUIRED C HEADER	REQUIRED C++ HEADER
asinh, **asinhf**, **asinhl**	<math.h>	<cmath> or <math.h<

For additional compatibility information, see Compatibility.

Example

```
// crt_asinh.c
// Compile by using: cl /W4 crt_asinh.c
// This program displays the hyperbolic sine of pi / 4
// and the arc hyperbolic sine of the result.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = sinh( pi / 4 );
   y = asinh( x );
   printf( "sinh( %f ) = %f\n", pi/4, x );
   printf( "asinh( %f ) = %f\n", x, y );
}
```

```
sinh( 0.785398 ) = 0.868671
asinh( 0.868671 ) = 0.785398
```

See also

Floating-Point Support
acosh, acoshf, acoshl
atanh, atanhf, atanhl
cosh, coshf, coshl
sinh, sinhf, sinhl
tanh, tanhf, tanhl

assert Macro, _assert, _wassert

10/31/2018 • 4 minutes to read • Edit Online

Evaluates an expression and, when the result is **false**, prints a diagnostic message and aborts the program.

Syntax

```
assert(
   expression
);
void _assert(
   char const* message,
   char const* filename,
   unsigned line
);
void _wassert(
   wchar_t const* message,
   wchar_t const* filename,
   unsigned line
);
```

Parameters

expression
A scalar expression (including pointer expressions) that evaluates to nonzero (**true**) or 0 (**false**).

message
The message to display.

filename
The name of the source file the assertion failed in.

line
The line number in the source file of the failed assertion.

Remarks

The **assert** macro is typically used to identify logic errors during program development. Use it to stop program execution when unexpected conditions occur by implementing the *expression* argument to evaluate to **false** only when the program is operating incorrectly. Assertion checks can be turned off at compile time by defining the macro **NDEBUG**. You can turn off the **assert** macro without modifying your source files by using a **/DNDEBUG** command-line option. You can turn off the **assert** macro in your source code by using a `#define NDEBUG` directive before <assert.h> is included.

The **assert** macro prints a diagnostic message when *expression* evaluates to **false** (0) and calls abort to terminate program execution. No action is taken if *expression* is **true** (nonzero). The diagnostic message includes the failed expression, the name of the source file and line number where the assertion failed.

The diagnostic message is printed in wide characters. Thus, it will work as expected even if there are Unicode characters in the expression.

The destination of the diagnostic message depends on the type of application that called the routine. Console applications always receive the message through **stderr**. In a Windows-based application, **assert** calls the Windows MessageBox function to create a message box to display the message along with an **OK** button. When

the user clicks **OK**, the program aborts immediately.

When the application is linked with a debug version of the run-time libraries, **assert** creates a message box with three buttons: **Abort**, **Retry**, and **Ignore**. If the user clicks **Abort**, the program aborts immediately. If the user clicks **Retry**, the debugger is called and the user can debug the program if just-in-time (JIT) debugging is enabled. If the user clicks **Ignore**, **assert** continues with its normal execution: creating the message box with the **OK** button. Note that clicking **Ignore** when an error condition exists can result in undefined behavior.

For more information about CRT debugging, see CRT Debugging Techniques.

The **_assert** and **_wassert** functions are internal CRT functions. They help minimize the code required in your object files to support assertions. We do not recommend that you call these functions directly.

The **assert** macro is enabled in both the release and debug versions of the C run-time libraries when **NDEBUG** is not defined. When **NDEBUG** is defined, the macro is available but does not evaluate its argument and has no effect. When it is enabled, the **assert** macro calls **_wassert** for its implementation. Other assertion macros, _ASSERT, _ASSERTE and _ASSERT_EXPR, are also available, but they only evaluate the expressions passed to them when the _DEBUG macro has been defined and when they are in code linked with the debug version of the C run-time libraries.

Requirements

ROUTINE	REQUIRED HEADER
assert, **_wassert**	<assert.h>

The signature of the **_assert** function is not available in a header file. The signature of the **_wassert** function is only available when the **NDEBUG** macro is not defined.

Example

In this program, the **analyze_string** function uses the **assert** macro to test several conditions related to string and length. If any of the conditions fails, the program prints a message indicating what caused the failure.

```
// crt_assert.c
// compile by using: cl /W4 crt_assert.c
#include <stdio.h>
#include <assert.h>
#include <string.h>

void analyze_string( char *string );   // Prototype

int main( void )
{
   char  test1[] = "abc", *test2 = NULL, test3[] = "";

   printf ( "Analyzing string '%s'\n", test1 ); fflush( stdout );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 ); fflush( stdout );
   analyze_string( test2 );
   printf ( "Analyzing string '%s'\n", test3 ); fflush( stdout );
   analyze_string( test3 );
}

// Tests a string to see if it is NULL,
// empty, or longer than 0 characters.
void analyze_string( char * string )
{
   assert( string != NULL );        // Cannot be NULL
   assert( *string != '\0' );       // Cannot be empty
   assert( strlen( string ) > 2 );  // Length must exceed 2
}
```

The program generates this output:

```
Analyzing string 'abc'
Analyzing string '(null)'
Assertion failed: string != NULL, file crt_assert.c, line 25
```

After the assertion failure, depending on the version of the operating system and run-time library, you may see a message box that contains something like the following:

```
A problem caused the program to stop working correctly. Windows will close the program and notify you if a
solution is available.
```

If a debugger is installed, choose the **Debug** button to start the debugger, or **Close program** to exit.

See also

Error Handling
Process and Environment Control
abort
raise
signal
_ASSERT, _ASSERTE, _ASSERT_EXPR Macros
_DEBUG

_ASSERT, _ASSERTE, _ASSERT_EXPR Macros

10/31/2018 • 4 minutes to read • Edit Online

Evaluate an expression and generate a debug report when the result is **False** (debug version only).

Syntax

```
// Typical usage:
_ASSERT_EXPR( booleanExpression, message );
_ASSERT( booleanExpression );
_ASSERTE( booleanExpression );
```

Parameters

booleanExpression
A scalar expression (including pointer expressions) that evaluates to nonzero (true) or 0 (false).

message
A wide string to display as part of the report.

Remarks

The **_ASSERT_EXPR**, **_ASSERT** and **_ASSERTE** macros provide an application with a clean and simple mechanism for checking assumptions during the debugging process. They are very flexible because they do not need to be enclosed in `#ifdef` statements to prevent them from being called in a retail build of an application. This flexibility is achieved by using the _DEBUG macro. **_ASSERT_EXPR**, **_ASSERT** and **_ASSERTE** are only available when **_DEBUG** is defined at compile time. When **_DEBUG** is not defined, calls to these macros are removed during preprocessing.

_ASSERT_EXPR, **_ASSERT** and **_ASSERTE** evaluate their *booleanExpression* argument and when the result is **false** (0), they print a diagnostic message and call _CrtDbgReportW to generate a debug report. The **_ASSERT** macro prints a simple diagnostic message, **_ASSERTE** includes a string representation of the failed expression in the message, and **_ASSERT_EXPR** includes the *message* string in the diagnostic message. These macros do nothing when *booleanExpression* evaluates to nonzero.

_ASSERT_EXPR, **_ASSERT** and **_ASSERTE** invoke **_CrtDbgReportW**, which causes all output to be in wide characters. **_ASSERTE** properly prints Unicode characters in *booleanExpression* and **_ASSERT_EXPR** prints Unicode characters in *message*.

Because the **_ASSERTE** macro specifies the failed expression, and **_ASSERT_EXPR** lets you specify a message in the generated report, they enable users to identify the problem without referring to the application source code. However, a disadvantage exists in that every *message* printed by **_ASSERT_EXPR** and every expression evaluated by **_ASSERTE** is included in the output (debug version) file of your application as a string constant. Therefore, if a large number of calls are made to **_ASSERT_EXPR** or **_ASSERTE**, these expressions can greatly increase the size of your output file.

Unless you specify otherwise with the _CrtSetReportMode and _CrtSetReportFile functions, messages appear in a pop-up dialog box equivalent to setting:

```
_CrtSetReportMode(CRT_ASSERT, _CRTDBG_MODE_WNDW);
```

_CrtDbgReportW generates the debug report and determines its destination or destinations, based on the current report mode or modes and file defined for the **_CRT_ASSERT** report type. By default, assertion failures and errors are directed to a debug message window. The _CrtSetReportMode and _CrtSetReportFile functions are used to define the destinations for each report type.

When the destination is a debug message window and the user clicks the **Retry** button, **_CrtDbgReportW** returns 1, causing the **_ASSERT_EXPR**, **_ASSERT** and **_ASSERTE** macros to start the debugger provided that just-in-time (JIT) debugging is enabled.

For more information about the reporting process, see the _CrtDbgReport, _CrtDbgReportW function. For more information about resolving assertion failures and using these macros as a debugging error handling mechanism, see Using Macros for Verification and Reporting.

In addition to the **_ASSERT** macros, the assert macro can be used to verify program logic. This macro is available in both the debug and release versions of the libraries. The _RPT, _RPTF debug macros are also available for generating a debug report, but they do not evaluate an expression. The **_RPT** macros generate a simple report. The **_RPTF** macros include the source file and line number where the report macro was called in the generated report. Wide character versions of these macros are available (**_RPTW**, **_RPTFW**). The wide character versions are identical to the narrow character versions except that wide character strings are used for all string parameters and output.

Although **_ASSERT_EXPR**, **_ASSERT** and **_ASSERTE** are macros and are available by including <crtdbg.h>, the application must link with a debug version of the C run-time library when **_DEBUG** is defined because these macros call other run-time functions.

Requirements

MACRO	REQUIRED HEADER
_ASSERT_EXPR, **_ASSERT**, **_ASSERTE**	<crtdbg.h>

Example

In this program, calls are made to the **_ASSERT** and **_ASSERTE** macros to test the condition `string1 == string2`. If the condition fails, these macros print a diagnostic message. The **_RPT** and **_RPTF** group of macros is also exercised in this program, as an alternative to the **printf** function.

```
// crt_ASSERT_macro.c
// compile with: /D_DEBUG /MTd /Od /Zi /link /verbose:lib /debug
//
// This program uses the _ASSERT and _ASSERTE debugging macros.
//

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <crtdbg.h>

int main()
{
   char *p1, *p2;

   // The Reporting Mode and File must be specified
   // before generating a debug report via an assert
   // or report macro.
   // This program sends all report types to STDOUT.
   _CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_WARN, _CRTDBG_FILE_STDOUT);
   _CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_ERROR, _CRTDBG_FILE_STDOUT);
   _CrtSetReportMode(_CRT_ASSERT, _CRTDBG_MODE_FILE);
   _CrtSetReportFile(_CRT_ASSERT, _CRTDBG_FILE_STDOUT);

   // Allocate and assign the pointer variables.
   p1 = (char *)malloc(10);
   strcpy_s(p1, 10, "I am p1");
   p2 = (char *)malloc(10);
   strcpy_s(p2, 10, "I am p2");

   // Use the report macros as a debugging
   // warning mechanism, similar to printf.
   // Use the assert macros to check if the
   // p1 and p2 variables are equivalent.
   // If the expression fails, _ASSERTE will
   // include a string representation of the
   // failed expression in the report.
   // _ASSERT does not include the
   // expression in the generated report.
   _RPT0(_CRT_WARN,
       "Use the assert macros to evaluate the expression p1 == p2.\n");
   _RPTF2(_CRT_WARN, "\n Will _ASSERT find '%s' == '%s' ?\n", p1, p2);
   _ASSERT(p1 == p2);

   _RPTF2(_CRT_WARN, "\n\n Will _ASSERTE find '%s' == '%s' ?\n",
          p1, p2);
   _ASSERTE(p1 == p2);

   _RPT2(_CRT_ERROR, "'%s' != '%s'\n", p1, p2);

   free(p2);
   free(p1);

   return 0;
}
```

```
Use the assert macros to evaluate the expression p1 == p2.
crt_ASSERT_macro.c(54) :
Will _ASSERT find 'I am p1' == 'I am p2' ?
crt_ASSERT_macro.c(55) : Assertion failed!
crt_ASSERT_macro.c(58) :

Will _ASSERTE find 'I am p1' == 'I am p2' ?
crt_ASSERT_macro.c(59) : Assertion failed: p1 == p2
'I am p1' != 'I am p2'
```

See also

Debug Routines
assert Macro, _assert, _wassert
_RPT, _RPTF, _RPTW, _RPTFW Macros

atan, atanf, atanl, atan2, atan2f, atan2l

10/31/2018 • 2 minutes to read • Edit Online

Calculates the arctangent of **x** (**atan**, **atanf**, and **atanl**) or the arctangent of **y**/**x** (**atan2**, **atan2f**, and **atan2l**).

Syntax

```
double atan( double x );
float atanf( float x );
long double atanl( long double x );

double atan2( double y, double x );
float atan2f( float y, float x );
long double atan2l( long double y, long double x );
```

```
float atan( float x );  // C++ only
long double atan( long double x );  // C++ only

float atan2( float y, float x );  // C++ only
long double atan2( long double y, long double x );  // C++ only
```

Parameters

x, *y*
Any numbers.

Return Value

atan returns the arctangent of *x* in the range -π/2 to π/2 radians. **atan2** returns the arctangent of *y*/*x* in the range -π to π radians. If *x* is 0, **atan** returns 0. If both parameters of **atan2** are 0, the function returns 0. All results are in radians.

atan2 uses the signs of both parameters to determine the quadrant of the return value.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	none	**_DOMAIN**

Remarks

The **atan** function calculates the arctangent (the inverse tangent function) of *x*. **atan2** calculates the arctangent of *y*/*x* (if *x* equals 0, **atan2** returns π/2 if *y* is positive, -π/2 if *y* is negative, or 0 if *y* is 0.)

atan has an implementation that uses Streaming SIMD Extensions 2 (SSE2). For information and restrictions about using the SSE2 implementation, see _set_SSE2_enable.

Because C++ allows overloading, you can call overloads of **atan** and **atan2** that take **float** or **long double** arguments. In a C program, **atan** and **atan2** always take **double** arguments and return a **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
atan, **atan2**, **atanf**, **atan2f**, **atanl**, **atan2l**	<math.h>	<cmath> or <math.h>

Example

```
// crt_atan.c
// arguments: 5 0.5
#include <math.h>
#include <stdio.h>
#include <errno.h>

int main( int ac, char* av[] )
{
   double x, y, theta;
   if( ac != 3 ){
      fprintf( stderr, "Usage: %s <x> <y>\n", av[0] );
      return 1;
   }
   x = atof( av[1] );
   theta = atan( x );
   printf( "Arctangent of %f: %f\n", x, theta );
   y = atof( av[2] );
   theta = atan2( y, x );
   printf( "Arctangent of %f / %f: %f\n", y, x, theta );
   return 0;
}
```

```
Arctangent of 5.000000: 1.373401
Arctangent of 0.500000 / 5.000000: 0.099669
```

See also

Floating-Point Support
acos, acosf, acosl
asin, asinf, asinl
cos, cosf, cosl
_matherr
sin, sinf, sinl
tan, tanf, tanl
_CIatan
_CIatan2

atanh, atanhf, atanhl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the inverse hyperbolic tangent.

Syntax

```
double atanh( double x );
float atanhf( float x );
long double atanhl( long double x );
```

```
float atanh( float x );  // C++ only
long double atanh( long double x );  // C++ only
```

Parameters

x
Floating-point value.

Return Value

The **atanh** functions return the inverse hyberbolic tangent (arc hyperbolic tangent) of *x*. If *x* is greater than 1, or less than -1, **errno** is set to **EDOM** and the result is a quiet NaN. If *x* is equal to 1 or -1, a positive or negative infinity is returned, respectively, and **errno** is set to **ERANGE**.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	none	none
X ≥ 1; *x* ≤ -1	none	none

Remarks

Because C++ allows overloading, you can call overloads of **atanh** that take and return **float** or **long double** values. In a C program, **atanh** always takes and returns **double**.

Requirements

FUNCTION	C HEADER	C++ HEADER
atanh, **atanhf**, **atanhl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_atanh.c
// This program displays the hyperbolic tangent of pi / 4
// and the arc hyperbolic tangent of the result.
//

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = tanh( pi / 4 );
   y = atanh( x );
   printf( "tanh( %f ) = %f\n", pi/4, x );
   printf( "atanh( %f ) = %f\n", x, y );
}
```

```
tanh( 0.785398 ) = 0.655794
atanh( 0.655794 ) = 0.785398
```

See also

Floating-Point Support
acosh, acoshf, acoshl
asinh, asinhf, asinhl
cosh, coshf, coshl
sinh, sinhf, sinhl
tanh, tanhf, tanhl

atexit

10/31/2018 • 2 minutes to read • Edit Online

Processes the specified function at exit.

Syntax

```
int atexit(
   void (__cdecl *func )( void )
);
```

Parameters

func
Function to be called.

Return Value

atexit returns 0 if successful, or a nonzero value if an error occurs.

Remarks

The **atexit** function is passed the address of a function *func* to be called when the program terminates normally. Successive calls to **atexit** create a register of functions that are executed in last-in, first-out (LIFO) order. The functions passed to **atexit** cannot take parameters. **atexit** and **_onexit** use the heap to hold the register of functions. Thus, the number of functions that can be registered is limited only by heap memory.

The code in the **atexit** function should not contain any dependency on any DLL which could have already been unloaded when the **atexit** function is called.

To generate an ANSI-compliant application, use the ANSI-standard **atexit** function (rather than the similar **_onexit** function).

Requirements

ROUTINE	REQUIRED HEADER
atexit	<stdlib.h>

Example

This program pushes four functions onto the stack of functions to be executed when **atexit** is called. When the program exits, these programs are executed on a last in, first out basis.

```
// crt_atexit.c
#include <stdlib.h>
#include <stdio.h>

void fn1( void ), fn2( void ), fn3( void ), fn4( void );

int main( void )
{
   atexit( fn1 );
   atexit( fn2 );
   atexit( fn3 );
   atexit( fn4 );
   printf( "This is executed first.\n" );
}

void fn1()
{
   printf( "next.\n" );
}

void fn2()
{
   printf( "executed " );
}

void fn3()
{
   printf( "is " );
}

void fn4()
{
   printf( "This " );
}
```

```
This is executed first.
This is executed next.
```

See also

Process and Environment Control
abort
exit, _Exit, _exit
_onexit, _onexit_m

_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a string to a double (**_atodbl**), long double (**_atoldbl**), or float (**_atoflt**).

Syntax

```
int _atodbl( _CRT_DOUBLE * value, char * str );
int _atodbl_l ( _CRT_DOUBLE * value, char * str, locale_t locale );
int _atoldbl( _LDOUBLE * value, char * str );
int _atoldbl_l ( _LDOUBLE * value, char * str, locale_t locale );
int _atoflt( _CRT_FLOAT * value, const char * str );
int _atoflt_l( _CRT_FLOAT * value, const char * str, locale_t locale );
```

Parameters

value
The double, long double, or float value that's produced by converting the string to a floating-point value. These values are wrapped in a structure.

str
The string to be parsed to convert into a floating-point value.

locale
The locale to use.

Return Value

Returns 0 if successful. Possible error codes are **_UNDERFLOW** or **_OVERFLOW**, which are defined in the header file <math.h>.

Remarks

These functions convert a string to a floating-point value. The difference between these functions and the **atof** family of functions is that these functions do not generate floating-point code and do not cause hardware exceptions. Instead, error conditions are reported as error codes.

If a string does not have a valid interpretation as a floating-point value, *value* is set to zero and the return value is zero.

The versions of these functions that have the **_l** suffix are identical the versions that don't have the suffix, except that they use the *locale* parameter that's passed in instead of the current thread locale.

Requirements

ROUTINES	REQUIRED HEADER
_atodbl, **_atoldbl**, **_atoflt** **_atodbl_l**, **_atoldbl_l**, **_atoflt_l**	<stdlib.h>

Example

```
// crt_atodbl.c
// Uses _atodbl to convert a string to a double precision
// floating point value.

#include <stdlib.h>
#include <stdio.h>

int main()
{
   char str1[256] = "3.141592654";
   char abc[256] = "abc";
   char oflow[256] = "1.0E+5000";
   _CRT_DOUBLE dblval;
   _CRT_FLOAT fltval;
   int retval;

   retval = _atodbl(&dblval, str1);

   printf("Double value: %lf\n", dblval.x);
   printf("Return value: %d\n\n", retval);

   retval = _atoflt(&fltval, str1);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   // A non-floating point value: returns 0.
   retval = _atoflt(&fltval, abc);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   // Overflow.
   retval = _atoflt(&fltval, oflow);
   printf("Float value: %f\n", fltval.f);
   printf("Return value: %d\n\n", retval);

   return 0;
}
```

```
Double value: 3.141593
Return value: 0

Float value: 3.141593
Return value: 0

Float value: 0.000000
Return value: 0

Float value: inf
Return value: 3
```

See also

Data Conversion
Floating-Point Support
Locale
atof, _atof_l, _wtof, _wtof_l

atof, _atof_l, _wtof, _wtof_l

10/31/2018 • 2 minutes to read • Edit Online

Convert a string to double.

Syntax

```
double atof(
   const char *str
);
double _atof_l(
   const char *str,
   _locale_t locale
);
double _wtof(
   const wchar_t *str
);
double _wtof_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str
String to be converted.

locale
Locale to use.

Return Value

Each function returns the **double** value produced by interpreting the input characters as a number. The return value is 0.0 if the input cannot be converted to a value of that type.

In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a double-precision, floating-point value.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **atof** and **_wtof** has the following form:

[*whitespace*] [*sign*] [*digits*] [**.***digits*] [{**e** | **E** }[*sign*]*digits*]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, which consists of an introductory

letter (**e**, or **E**) and an optionally signed decimal integer.

The UCRT versions of these functions do not support conversion of Fortran-style (**d** or **D**) exponent letters. This non-standard extension was supported by earlier versions of the CRT, and may be a breaking change for your code.

The versions of these functions with the **_l** suffix are identical except that they use the *locale* parameter passed in instead of the current locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstof	**atof**	**atof**	**_wtof**
_ttof	**atof**	**atof**	**_wtof**

Requirements

ROUTINE(S)	REQUIRED HEADER
atof, **_atof_l**	C: <math.h> or <stdlib.h> C++: <cstdlib>, <stdlib.h>, <cmath> or <math.h>
_wtof, **_wtof_l**	C: <stdlib.h> or <wchar.h> C++: <cstdlib>, <stdlib.h> or <wchar.h>

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atof** and **_atof_l** functions.

```
// crt_atof.c
//
// This program shows how numbers stored as
// strings can be converted to numeric
// values using the atof and _atof_l functions.

#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main(void)
{
    char    *str = NULL;
    double value = 0;
    _locale_t fr = _create_locale(LC_NUMERIC, "fr-FR");

    // An example of the atof function
    // using leading and training spaces.
    str = "  3336402735171707160320 ";
    value = atof(str);
    printf("Function: atof(\"%s\") = %e\n", str, value);

    // Another example of the atof function
    // using the 'E' exponential formatting keyword.
    str = "3.1412764583E210";
    value = atof(str);
    printf("Function: atof(\"%s\") = %e\n", str, value);

    // An example of the atof and _atof_l functions
    // using the 'e' exponential formatting keyword
    // and showing different decimal point interpretations.
    str = "  -2,309e-25";
    value = atof(str);
    printf("Function: atof(\"%s\") = %e\n", str, value);
    value = _atof_l(str, fr);
    printf("Function: _atof_l(\"%s\", fr)) = %e\n", str, value);
}
```

```
Function: atof("  3336402735171707160320 ") = 3.336403e+21
Function: atof("3.1412764583E210") = 3.141276e+210
Function: atof("  -2,309e-25") = -2.000000e+00
Function: _atof_l("  -2,309e-25", fr)) = -2.309000e-25
```

See also

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

atoi, _atoi_l, _wtoi, _wtoi_l

3/1/2019 • 2 minutes to read • Edit Online

Convert a string to integer.

Syntax

```
int atoi(
   const char *str
);
int _wtoi(
   const wchar_t *str
);
int _atoi_l(
   const char *str,
   _locale_t locale
);
int _wtoi_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str
String to be converted.

locale
Locale to use.

Return Value

Each function returns the **int** value produced by interpreting the input characters as a number. The return value is 0 for **atoi** and **_wtoi**, if the input cannot be converted to a value of that type.

In the case of overflow with large negative integral values, **LONG_MIN** is returned. **atoi** and **_wtoi** return **INT_MAX** and **INT_MIN** on these conditions. In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to an integer value (**atoi** and **_wtoi**). The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **atoi** and **_wtoi** has the following form:

```
[whitespace] [sign] [digits]]
```

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more digits.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstoi	**atoi**	**atoi**	**_wtoi**
_ttoi	**atoi**	**atoi**	**_wtoi**

Requirements

ROUTINES	REQUIRED HEADER
atoi	<stdlib.h>
_atoi_l, **_wtoi**, **_wtoi_l**	<stdlib.h> or <wchar.h>

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atoi** functions.

```
// crt_atoi.c
// This program shows how numbers
// stored as strings can be converted to
// numeric values using the atoi functions.

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    int     value = 0;

    // An example of the atoi function.
    str = "  -2309 ";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );

    // Another example of the atoi function.
    str = "31412764";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );

    // Another example of the atoi function
    // with an overflow condition occurring.
    str = "3336402735171707160320";
    value = atoi( str );
    printf( "Function: atoi( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

```
Function: atoi( "  -2309 " ) = -2309
Function: atoi( "31412764" ) = 31412764
Function: atoi( "3336402735171707160320" ) = 2147483647
Overflow condition occurred.
```

See also

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

_atoi64, _atoi64_l, _wtoi64, _wtoi64_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a string to a 64-bit integer.

Syntax

```
__int64 _atoi64(
   const char *str
);
__int64 _wtoi64(
   const wchar_t *str
);
__int64 _atoi64_l(
   const char *str,
   _locale_t locale
);
__int64 _wtoi64_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str
String to be converted.

locale
Locale to use.

Return Value

Each function returns the **__int64** value produced by interpreting the input characters as a number. The return value is 0 for **_atoi64** if the input cannot be converted to a value of that type.

In the case of overflow with large positive integral values, **_atoi64** returns **I64_MAX** and **I64_MIN** in the case of overflow with large negative integral values.

In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a 64-bit integer value.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character might be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **_atoi64** has the following form:

[*whitespace*] [*sign*] [*digits*]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more digits.

_wtoi64 is identical to **_atoi64** except that it takes a wide character string as a parameter.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstoi64	**_atoi64**	**_atoi64**	**_wtoi64**
_ttoi64	**_atoi64**	**_atoi64**	**_wtoi64**

Requirements

ROUTINES	REQUIRED HEADER
_atoi64, **_atoi64_l**	<stdlib.h>
_wtoi64, **_wtoi64_l**	<stdlib.h> or <wchar.h>

Example

This program shows how numbers stored as strings can be converted to numeric values using the **_atoi64** functions.

```
// crt_atoi64.c
// This program shows how numbers stored as
// strings can be converted to numeric values
// using the _atoi64 functions.
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    __int64 value = 0;

    // An example of the _atoi64 function
    // with leading and trailing white spaces.
    str = "  -2309 ";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );

    // Another example of the _atoi64 function
    // with an arbitrary decimal point.
    str = "314127.64";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );

    // Another example of the _atoi64 function
    // with an overflow condition occurring.
    str = "333640273517170716032 0";
    value = _atoi64( str );
    printf( "Function: _atoi64( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

```
Function: _atoi64( "  -2309 " ) = -2309
Function: _atoi64( "314127.64" ) = 314127
Function: _atoi64( "333640273517170716032 0" ) = -1
Overflow condition occurred.
```

See also

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

atol, _atol_l, _wtol, _wtol_l

10/31/2018 • 2 minutes to read • Edit Online

Convert a string to a long integer.

Syntax

```
long atol(
   const char *str
);
long _atol_l(
   const char *str,
   _locale_t locale
);
long _wtol(
   const wchar_t *str
);
long _wtol_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str
String to be converted.

locale
Locale to use.

Return Value

Each function returns the **long** value produced by interpreting the input characters as a number. The return value is 0L for **atol** if the input cannot be converted to a value of that type.

In the case of overflow with large positive integral values, **atol** returns **LONG_MAX**; in the case of overflow with large negative integral values, **LONG_MIN** is returned. In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a long integer value (**atol**).

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0' or L'\0') terminating the string.

The *str* argument to **atol** has the following form:

> [*whitespace*] [*sign*] [*digits*]]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more digits.

_wtol is identical to **atol** except that it takes a wide character string.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstol	**atol**	**atol**	**_wtol**
_ttol	**atol**	**atol**	**_wtol**

Requirements

ROUTINES	REQUIRED HEADER
atol	<stdlib.h>
_atol_l, **_wtol**, **_wtol_l**	<stdlib.h> and <wchar.h>

Example

This program shows how numbers stored as strings can be converted to numeric values using the **atol** function.

```
// crt_atol.c
// This program shows how numbers stored as
// strings can be converted to numeric values
// using the atol functions.
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
    char    *str = NULL;
    long    value = 0;

    // An example of the atol function
    // with leading and trailing white spaces.
    str = "  -2309 ";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );

    // Another example of the atol function
    // with an arbitrary decimal point.
    str = "314127.64";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );

    // Another example of the atol function
    // with an overflow condition occurring.
    str = "3336402735171707160320";
    value = atol( str );
    printf( "Function: atol( \"%s\" ) = %d\n", str, value );
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

```
Function: atol( "  -2309 " ) = -2309
Function: atol( "314127.64" ) = 314127
Function: atol( "3336402735171707160320" ) = 2147483647
Overflow condition occurred.
```

See also

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

atoll, _atoll_l, _wtoll, _wtoll_l

11/9/2018 • 2 minutes to read • Edit Online

Converts a string to a **long long** integer.

Syntax

```
long long atoll(
   const char *str
);
long long _wtoll(
   const wchar_t *str
);
long long _atoll_l(
   const char *str,
   _locale_t locale
);
long long _wtoll_l(
   const wchar_t *str,
   _locale_t locale
);
```

Parameters

str
String to be converted.

locale
Locale to use.

Return Value

Each function returns the **long long** value that's produced by interpreting the input characters as a number. The return value for **atoll** is 0 if the input cannot be converted to a value of that type.

For overflow with large positive integral values, **atoll** returns **LLONG_MAX**, and for overflow with large negative integral values, it returns **LLONG_MIN**.

In all out-of-range cases, **errno** is set to **ERANGE**. If the parameter that's passed in is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return 0.

Remarks

These functions convert a character string to a **long long** integer value.

The input string is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character might be the null character ('\0' or L'\0') that terminates the string.

The *str* argument to **atoll** has the following form:

> [*whitespace*] [*sign*] [*digits*]

A *whitespace* consists of space or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more digits.

_wtoll is identical to **atoll** except that it takes a wide character string as a parameter.

The versions of these functions that have the **_l** suffix are identical to the versions that don't have it, except that they use the locale parameter that's passed in instead of the current locale. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstoll	**atoll**	**atoll**	**_wtoll**
_tstoll_l	**_atoll_l**	**_atoll_l**	**_wtoll_l**
_ttoll	**_atoll**	**_atoll**	**_wtoll**

Requirements

ROUTINES	REQUIRED HEADER
atoll, **_atoll_l**	<stdlib.h>
_wtoll, **_wtoll_l**	<stdlib.h> or <wchar.h>

Example

This program shows how to use the **atoll** functions to convert numbers stored as strings to numeric values.

```
// crt_atoll.c
// Build with: cl /W4 /Tc crt_atoll.c
// This program shows how to use the atoll
// functions to convert numbers stored as
// strings to numeric values.
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main(void)
{
    char *str = NULL;
    long long value = 0;

    // An example of the atoll function
    // with leading and trailing white spaces.
    str = "  -27182818284 ";
    value = atoll(str);
    printf("Function: atoll(\"%s\") = %lld\n", str, value);

    // Another example of the atoll function
    // with an arbitrary decimal point.
    str = "314127.64";
    value = atoll(str);
    printf("Function: atoll(\"%s\") = %lld\n", str, value);

    // Another example of the atoll function
    // with an overflow condition occurring.
    str = "333640273517170716032 0";
    value = atoll(str);
    printf("Function: atoll(\"%s\") = %lld\n", str, value);
    if (errno == ERANGE)
    {
       printf("Overflow condition occurred.\n");
    }
}
```

```
Function: atoll("  -27182818284 ") = -27182818284
Function: atoll("314127.64") = 314127
Function: atoll("333640273517170716032 0") = 9223372036854775807
Overflow condition occurred.
```

See also

Data Conversion
Floating-Point Support
Locale
_ecvt
_fcvt
_gcvt
setlocale, _wsetlocale
_atodbl, _atodbl_l, _atoldbl, _atoldbl_l, _atoflt, _atoflt_l

_beginthread, _beginthreadex

1/24/2019 • 9 minutes to read • Edit Online

Creates a thread.

Syntax

```
uintptr_t _beginthread( // NATIVE CODE
   void( __cdecl *start_address )( void * ),
   unsigned stack_size,
   void *arglist
);
uintptr_t _beginthread( // MANAGED CODE
   void( __clrcall *start_address )( void * ),
   unsigned stack_size,
   void *arglist
);
uintptr_t _beginthreadex( // NATIVE CODE
   void *security,
   unsigned stack_size,
   unsigned ( __stdcall *start_address )( void * ),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr
);
uintptr_t _beginthreadex( // MANAGED CODE
   void *security,
   unsigned stack_size,
   unsigned ( __clrcall *start_address )( void * ),
   void *arglist,
   unsigned initflag,
   unsigned *thrdaddr
);
```

Parameters

start_address
Start address of a routine that begins execution of a new thread. For **_beginthread**, the calling convention is either __cdecl (for native code) or __clrcall (for managed code); for **_beginthreadex**, it is either __stdcall (for native code) or __clrcall (for managed code).

stack_size
Stack size for a new thread, or 0.

arglist
Argument list to be passed to a new thread, or **NULL**.

Security
Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *Security* is **NULL**, the handle cannot be inherited. Must be **NULL** for Windows 95 applications.

initflag
Flags that control the initial state of a new thread. Set *initflag* to 0 to run immediately, or to **CREATE_SUSPENDED** to create the thread in a suspended state; use ResumeThread to execute the thread. Set *initflag* to **STACK_SIZE_PARAM_IS_A_RESERVATION** flag to use *stack_size* as the initial reserve size of the stack in bytes; if this flag is not specified, *stack_size* specifies the commit size.

thrdaddr
Points to a 32-bit variable that receives the thread identifier. If it's **NULL**, it's not used.

Return Value

If successful, each of these functions returns a handle to the newly created thread; however, if the newly created thread exits too quickly, **_beginthread** might not return a valid handle. (See the discussion in the Remarks section.) On an error, **_beginthread** returns -1L, and **errno** is set to **EAGAIN** if there are too many threads, to **EINVAL** if the argument is invalid or the stack size is incorrect, or to **EACCES** if there are insufficient resources (such as memory). On an error, **_beginthreadex** returns 0, and **errno** and **_doserrno** are set.

If *start_address* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For more information about **uintptr_t**, see Standard Types.

Remarks

The **_beginthread** function creates a thread that begins execution of a routine at *start_address*. The routine at *start_address* must use the **__cdecl** (for native code) or **__clrcall** (for managed code) calling convention and should have no return value. When the thread returns from that routine, it is terminated automatically. For more information about threads, see Multithreading Support for Older Code (Visual C++).

_beginthreadex resembles the Win32 CreateThread API more closely than **_beginthread** does. **_beginthreadex** differs from **_beginthread** in the following ways:

- **_beginthreadex** has three additional parameters: *initflag*, *Security*, and **threadaddr**. The new thread can be created in a suspended state, with a specified security, and can be accessed by using *thrdaddr*, which is the thread identifier.
- The routine at *start_address* that's passed to **_beginthreadex** must use the **__stdcall** (for native code) or **__clrcall** (for managed code) calling convention and must return a thread exit code.
- **_beginthreadex** returns 0 on failure, rather than -1L.
- A thread that's created by using **_beginthreadex** is terminated by a call to _endthreadex.

The **_beginthreadex** function gives you more control over how the thread is created than **_beginthread** does. The **_endthreadex** function is also more flexible. For example, with **_beginthreadex**, you can use security information, set the initial state of the thread (running or suspended), and get the thread identifier of the newly created thread. You can also use the thread handle that's returned by **_beginthreadex** with the synchronization APIs, which you cannot do with **_beginthread**.

It's safer to use **_beginthreadex** than **_beginthread**. If the thread that's generated by **_beginthread** exits quickly, the handle that's returned to the caller of **_beginthread** might be invalid or point to another thread. However, the handle that's returned by **_beginthreadex** has to be closed by the caller of **_beginthreadex**, so it is guaranteed to be a valid handle if **_beginthreadex** did not return an error.

You can call _endthread or **_endthreadex** explicitly to terminate a thread; however, **_endthread** or **_endthreadex** is called automatically when the thread returns from the routine that's passed as a parameter. Terminating a thread with a call to **_endthread** or **_endthreadex** helps ensure correct recovery of resources that are allocated for the thread.

_endthread automatically closes the thread handle, whereas **_endthreadex** does not. Therefore, when you use **_beginthread** and **_endthread**, do not explicitly close the thread handle by calling the Win32 CloseHandle API. This behavior differs from the Win32 ExitThread API.

> **NOTE**
>
> For an executable file linked with Libcmt.lib, do not call the Win32 **ExitThread** API so that you don't prevent the run-time system from reclaiming allocated resources. **_endthread** and **_endthreadex** reclaim allocated thread resources and then call **ExitThread**.

The operating system handles the allocation of the stack when either **_beginthread** or **_beginthreadex** is called; you don't have to pass the address of the thread stack to either of these functions. In addition, the *stack_size* argument can be 0, in which case the operating system uses the same value as the stack that's specified for the main thread.

arglist is a parameter to be passed to the newly created thread. Typically, it is the address of a data item, such as a character string. *arglist* can be **NULL** if it is not needed, but **_beginthread** and **_beginthreadex** must be given some value to pass to the new thread. All threads are terminated if any thread calls abort, **exit**, **_exit**, or **ExitProcess**.

The locale of the new thread is initialized by using the per-process global current locale info. If per-thread locale is enabled by a call to _configthreadlocale (either globally or for new threads only), the thread can change its locale independently from other threads by calling **setlocale** or **_wsetlocale**. Threads that don't have the per-thread locale flag set can affect the locale info in all other threads that also don't have the per-thread locale flag set, as well as all newly-created threads. For more information, see Locale.

For **/clr** code, **_beginthread** and **_beginthreadex** each have two overloads. One takes a native calling-convention function pointer, and the other takes a **__clrcall** function pointer. The first overload is not application domain-safe and never will be. If you are writing **/clr** code you must ensure that the new thread enters the correct application domain before it accesses managed resources. You can do this, for example, by using call_in_appdomain Function. The second overload is application domain-safe; the newly created thread will always end up in the application domain of the caller of **_beginthread** or **_beginthreadex**.

Requirements

ROUTINE	REQUIRED HEADER
_beginthread	<process.h>
_beginthreadex	<process.h>

For more compatibility information, see Compatibility.

Libraries

Multithreaded versions of the C run-time libraries only.

To use **_beginthread** or **_beginthreadex**, the application must link with one of the multithreaded C run-time libraries.

Example

The following example uses **_beginthread** and **_endthread**.

```
// crt_BEGTHRD.C
// compile with: /MT /D "_X86_" /c
// processor: x86
#include <windows.h>
#include <process.h>    /* _beginthread, _endthread */
```

```
#include <stddef.h>
#include <stdlib.h>
#include <conio.h>

void Bounce( void * );
void CheckKey( void * );

// GetRandom returns a random integer between min and max.
#define GetRandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))
// GetGlyph returns a printable ASCII character value
#define GetGlyph( val ) ((char)((val + 32) % 93 + 33))

BOOL repeat = TRUE;                 // Global repeat flag
HANDLE hStdOut;                     // Handle for console window
CONSOLE_SCREEN_BUFFER_INFO csbi;    // Console information structure

int main()
{
    int param = 0;
    int * pparam = &param;

    // Get display screen's text row and column information.
    hStdOut = GetStdHandle( STD_OUTPUT_HANDLE );
    GetConsoleScreenBufferInfo( hStdOut, &csbi );

    // Launch CheckKey thread to check for terminating keystroke.
    _beginthread( CheckKey, 0, NULL );

    // Loop until CheckKey terminates program or 1000 threads created.
    while( repeat && param < 1000 )
    {
        // launch another character thread.
        _beginthread( Bounce, 0, (void *) pparam );

        // increment the thread parameter
        param++;

        // Wait one second between loops.
        Sleep( 1000L );
    }
}

// CheckKey - Thread to wait for a keystroke, then clear repeat flag.
void CheckKey( void * ignored )
{
    _getch();
    repeat = 0;    // _endthread implied
}

// Bounce - Thread to create and control a colored letter that moves
// around on the screen.
//
// Params: parg - the value to create the character from
void Bounce( void * parg )
{
    char       blankcell = 0x20;
    CHAR_INFO  ci;
    COORD      oldcoord, cellsize, origin;
    DWORD      result;
    SMALL_RECT region;

    cellsize.X = cellsize.Y = 1;
    origin.X = origin.Y = 0;

    // Generate location, letter and color attribute from thread argument.
    srand( _threadid );
    oldcoord.X = region.Left = region.Right =
        GetRandom(csbi.srWindow.Left, csbi.srWindow.Right - 1);
    oldcoord.Y = region.Top = region.Bottom =
```

```
        GetRandom(csbi.srWindow.Top, csbi.srWindow.Bottom - 1);
    ci.Char.AsciiChar = GetGlyph(*((int *)parg));
    ci.Attributes = GetRandom(1, 15);

    while (repeat)
    {
        // Pause between loops.
        Sleep( 100L );

        // Blank out our old position on the screen, and draw new letter.
        WriteConsoleOutputCharacterA(hStdOut, &blankcell, 1, oldcoord, &result);
        WriteConsoleOutputA(hStdOut, &ci, cellsize, origin, &region);

        // Increment the coordinate for next placement of the block.
        oldcoord.X = region.Left;
        oldcoord.Y = region.Top;
        region.Left = region.Right += GetRandom(-1, 1);
        region.Top = region.Bottom += GetRandom(-1, 1);

        // Correct placement (and beep) if about to go off the screen.
        if (region.Left < csbi.srWindow.Left)
            region.Left = region.Right = csbi.srWindow.Left + 1;
        else if (region.Right >= csbi.srWindow.Right)
            region.Left = region.Right = csbi.srWindow.Right - 2;
        else if (region.Top < csbi.srWindow.Top)
            region.Top = region.Bottom = csbi.srWindow.Top + 1;
        else if (region.Bottom >= csbi.srWindow.Bottom)
            region.Top = region.Bottom = csbi.srWindow.Bottom - 2;

        // If not at a screen border, continue, otherwise beep.
        else
            continue;
        Beep((ci.Char.AsciiChar - 'A') * 100, 175);
    }
    // _endthread given to terminate
    _endthread();
}
```

Press any key to end the sample application.

Example

The following sample code demonstrates how you can use the thread handle that's returned by **_beginthreadex** with the synchronization API WaitForSingleObject. The main thread waits for the second thread to terminate before it continues. When the second thread calls **_endthreadex**, it causes its thread object to go to the signaled state. This allows the primary thread to continue running. This cannot be done with **_beginthread** and **_endthread**, because **_endthread** calls **CloseHandle**, which destroys the thread object before it can be set to the signaled state.

```
// crt_begthrdex.cpp
// compile with: /MT
#include <windows.h>
#include <stdio.h>
#include <process.h>

unsigned Counter;
unsigned __stdcall SecondThreadFunc( void* pArguments )
{
    printf( "In second thread...\n" );

    while ( Counter < 1000000 )
        Counter++;

    _endthreadex( 0 );
    return 0;
}

int main()
{
    HANDLE hThread;
    unsigned threadID;

    printf( "Creating second thread...\n" );

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc, NULL, 0, &threadID );

    // Wait until second thread terminates. If you comment out the line
    // below, Counter will not be correct because the thread has not
    // terminated, and Counter most likely has not been incremented to
    // 1000000 yet.
    WaitForSingleObject( hThread, INFINITE );
    printf( "Counter should be 1000000; it is-> %d\n", Counter );
    // Destroy the thread object.
    CloseHandle( hThread );
}
```

```
Creating second thread...
In second thread...
Counter should be 1000000; it is-> 1000000
```

See also

- Process and Environment Control
- _endthread, _endthreadex
- abort
- exit, _Exit, _exit
- GetExitCodeThread

Bessel Functions: _j0, _j1, _jn, _y0, _y1, _yn

10/31/2018 • 2 minutes to read • Edit Online

Computes the Bessel function of the first or second kind, of orders 0, 1, or n. The Bessel functions are commonly used in the mathematics of electromagnetic wave theory.

Syntax

```
double _j0(
   double x
);
double _j1(
   double x
);
double _jn(
   int n,
   double x
);
double _y0(
   double x
);
double _y1(
   double x
);
double _yn(
   int n,
   double x
);
```

Parameters

x
Floating-point value.

n
Integer order of Bessel function.

Return Value

Each of these routines returns a Bessel function of *x*. If *x* is negative in the **_y0**, **_y1**, or **_yn** functions, the routine sets **errno** to **EDOM**, prints a **_DOMAIN** error message to **stderr**, and returns **_HUGE_VAL**. You can modify error handling by using **_matherr**.

Remarks

The **_j0**, **_j1**, and **_jn** routines return Bessel functions of the first kind: orders 0, 1, and n, respectively.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	**INVALID**	**_DOMAIN**

The **_y0**, **_y1**, and **_yn** routines return Bessel functions of the second kind: orders 0, 1, and n, respectively.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	**INVALID**	**_DOMAIN**
± 0	**ZERODIVIDE**	**_SING**
|x| < 0.0	**INVALID**	**_DOMAIN**

Requirements

ROUTINE	REQUIRED HEADER
_j0, **_j1**, **_jn**, **_y0**, **_y1**, **_yn**	<cmath> (C++), <math.h> (C, C++)

For additional compatibility information, see Compatibility.

Example

```
// crt_bessel1.c
#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.387;
   int n = 3, c;

   printf( "Bessel functions for x = %f:\n", x );
   printf( "   Kind   Order  Function     Result\n\n" );
   printf( "   First  0      _j0( x )     %f\n", _j0( x ) );
   printf( "   First  1      _j1( x )     %f\n", _j1( x ) );
   for( c = 2; c < 5; c++ )
      printf( "   First  %d      _jn( %d, x )  %f\n", c, c, _jn( c, x ) );
   printf( "   Second 0      _y0( x )     %f\n", _y0( x ) );
   printf( "   Second 1      _y1( x )     %f\n", _y1( x ) );
   for( c = 2; c < 5; c++ )
      printf( "   Second %d      _yn( %d, x )  %f\n", c, c, _yn( c, x ) );
}
```

```
Bessel functions for x = 2.387000:
   Kind   Order  Function     Result

   First  0      _j0( x )     0.009288
   First  1      _j1( x )     0.522941
   First  2      _jn( 2, x )  0.428870
   First  3      _jn( 3, x )  0.195734
   First  4      _jn( 4, x )  0.063131
   Second 0      _y0( x )     0.511681
   Second 1      _y1( x )     0.094374
   Second 2      _yn( 2, x )  -0.432608
   Second 3      _yn( 3, x )  -0.819314
   Second 4      _yn( 4, x )  -1.626833
```

See also

Floating-Point Support
_matherr

bitand

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the & operator.

Syntax

```
#define bitand &
```

Remarks

The macro yields the operator

Example

```
// iso646_bitand.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, b = 2, result;

   result = a & b;
   cout << result << endl;

   result= a bitand b;
   cout << result << endl;
}
```

```
0
0
```

Requirements

Header: <iso646.h>

bitor

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the | operator.

Syntax

```
#define bitor |
```

Remarks

The macro yields the operator |.

Example

```
// iso646_bitor.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, b = 2, result;

   result = a | b;
   cout << result << endl;

   result= a bitor b;
   cout << result << endl;
}
```

```
3
3
```

Requirements

Header: <iso646.h>

bsearch

3/1/2019 • 2 minutes to read • Edit Online

Performs a binary search of a sorted array. A more secure version of this function is available; see bsearch_s.

Syntax

```
void *bsearch(
   const void *key,
   const void *base,
   size_t num,
   size_t width,
   int ( __cdecl *compare ) (const void *key, const void *datum)
);
```

Parameters

key
Object to search for.

base
Pointer to base of search data.

number
Number of elements.

width
Width of elements.

compare
Callback function that compares two elements. The first is a pointer to the key for the search and the second is a pointer to the array element to be compared with the key.

Return Value

bsearch returns a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**. If the array is not in ascending sort order or contains duplicate records with identical keys, the result is unpredictable.

Remarks

The **bsearch** function performs a binary search of a sorted array of *number* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought. The *compare* parameter is a pointer to a user-supplied routine that compares the requested key to an array element and returns one of the following values specifying their relationship:

VALUE RETURNED BY *COMPARE* ROUTINE	DESCRIPTION
< 0	Key is less than array element.
0	Key is equal to array element.

VALUE RETURNED BY *COMPARE* ROUTINE	DESCRIPTION
> 0	Key is greater than array element.

This function validates its parameters. If *compare*, *key* or *number* is **NULL**, or if *base* is **NULL** and *number* is nonzero, or if *width* is zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to `EINVAL` and the function returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
bsearch	<stdlib.h> and <search.h>

For additional compatibility information, see Compatibility.

Example

This program sorts a string array with qsort, and then uses bsearch to find the word "cat".

```
// crt_bsearch.c
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare( char **arg1, char **arg2 )
{
   /* Compare all of both strings: */
   return _strcmpi( *arg1, *arg2 );
}

int main( void )
{
   char *arr[] = {"dog", "pig", "horse", "cat", "human", "rat", "cow", "goat"};
   char **result;
   char *key = "cat";
   int i;

   /* Sort using Quicksort algorithm: */
   qsort( (void *)arr, sizeof(arr)/sizeof(arr[0]), sizeof( char * ), (int (*)(const
   void*, const void*))compare );

   for( i = 0; i < sizeof(arr)/sizeof(arr[0]); ++i )    /* Output sorted list */
      printf( "%s ", arr[i] );

   /* Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch( (char *) &key, (char *)arr, sizeof(arr)/sizeof(arr[0]),
                              sizeof( char * ), (int (*)(const void*, const void*))compare );
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}
```

```
cat cow dog goat horse human pig rat
cat found at 002F0F04
```

See also

Searching and Sorting
_lfind
_lsearch
qsort

bsearch_s

3/1/2019 • 3 minutes to read • Edit Online

Performs a binary search of a sorted array. This is version of bsearch with security enhancements as described in Security Features in the CRT.

Syntax

```
void *bsearch_s(
   const void *key,
   const void *base,
   size_t number,
   size_t width,
   int ( __cdecl *compare ) ( void *, const void *key, const void *datum),
   void * context
);
```

Parameters

key
Object to search for.

base
Pointer to base of search data.

number
Number of elements.

width
Width of elements.

compare
Callback function that compares two elements. The first argument is the *context* pointer. The second argument is a pointer to the *key* for the search. The third argument is a pointer to the array element to be compared with *key*.

context
A pointer to an object that can be accessed in the comparison function.

Return Value

bsearch_s returns a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**. If the array is not in ascending sort order or contains duplicate records with identical keys, the result is unpredictable.

If invalid parameters are passed to the function, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

key	*base*	*compare*	*number*	*width*	**errno**

NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	= 0	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **bsearch_s** function performs a binary search of a sorted array of *number* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought. The *compare* parameter is a pointer to a user-supplied routine that compares the requested key to an array element and returns one of the following values specifying their relationship:

VALUE RETURNED BY *COMPARE* ROUTINE	DESCRIPTION
< 0	Key is less than array element.
0	Key is equal to array element.
> 0	Key is greater than array element.

The *context* pointer may be useful if the searched data structure is part of an object, and the compare function needs to access members of the object. The *compare* function may cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* parameter makes **bsearch_s** more secure since additional context may be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

ROUTINE	REQUIRED HEADER
bsearch_s	<stdlib.h> and <search.h>

For additional compatibility information, see Compatibility.

Example

This program sorts a string array with qsort_s, and then uses bsearch_s to find the word "cat".

```
// crt_bsearch_s.cpp
// This program uses bsearch_s to search a string array,
// passing a locale as the context.
// compile with: /EHsc
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;
```

```
// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
#define ENGLISH_LOCALE "English_US.850"
#endif

#ifdef CODEPAGE_1252
#define ENGLISH_LOCALE "English_US.1252"
#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making it vulnerable to thread conflicts
// (if this were a multithreaded program).

int compare( void *pvlocale, char **str1, char **str2)
{
    char *s1 = *str1;
    char *s2 = *str2;

    locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

    return use_facet< collate<char> >(loc).compare(
       s1, s1+strlen(s1),
       s2, s2+strlen(s2) );
}

int main( void )
{
   char *arr[] = {"dog", "pig", "horse", "cat", "human", "rat", "cow", "goat"};

   char *key = "cat";
   char **result;
   int i;

   /* Sort using Quicksort algorithm: */
   qsort_s( arr,
            sizeof(arr)/sizeof(arr[0]),
            sizeof( char * ),
            (int (*)(void*, const void*, const void*))compare,
            &locale(ENGLISH_LOCALE) );

   for( i = 0; i < sizeof(arr)/sizeof(arr[0]); ++i )    /* Output sorted list */
      printf( "%s ", arr[i] );

   /* Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch_s( &key,
                                arr,
                                sizeof(arr)/sizeof(arr[0]),
                                sizeof( char * ),
                                (int (*)(void*, const void*, const void*))compare,
                                &locale(ENGLISH_LOCALE) );
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}
```

```
cat cow dog goat horse human pig rat
cat found at 002F0F04
```

See also

Searching and Sorting
_lfind
_lsearch
qsort

btowc

10/31/2018 • 2 minutes to read • Edit Online

Determine whether an integer represents a valid single-byte character in the initial shift state.

Syntax

```
wint_t btowc(
   int character
);
```

Parameters

character
Integer to test.

Return Value

Returns the wide-character representation of the character if the integer represents a valid single-byte character in the initial shift state. Returns WEOF if the integer is EOF or is not a valid single-byte character in the initial shift state. The output of this function is affected by the current **LC_TYPE** locale.

Requirements

ROUTINE	REQUIRED HEADER
btowc	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

mbtowc, _mbtowc_l

_byteswap_uint64, _byteswap_ulong, _byteswap_ushort

10/31/2018 • 2 minutes to read • Edit Online

Reverses the order of bytes in an integer.

Syntax

```
unsigned short _byteswap_ushort ( unsigned short val );
unsigned long _byteswap_ulong ( unsigned long val );
unsigned __int64 _byteswap_uint64 ( unsigned __int64 val );
```

Parameters

val
The integer to reverse byte order.

Requirements

ROUTINE	REQUIRED HEADER
_byteswap_ushort	<stdlib.h>
_byteswap_ulong	<stdlib.h>
_byteswap_uint64	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_byteswap.c
#include <stdlib.h>

int main()
{
   unsigned __int64 u64 = 0x0102030405060708;
   unsigned long ul = 0x01020304;

   printf("byteswap of %I64x = %I64x\n", u64, _byteswap_uint64(u64));
   printf("byteswap of %Ix = %Ix", ul, _byteswap_ulong(ul));
}
```

```
byteswap of 102030405060708 = 807060504030201
byteswap of 1020304 = 4030201
```

See also

Universal C runtime routines by category

c16rtomb, c32rtomb

2/4/2019 • 2 minutes to read • Edit Online

Convert a UTF-16 or UTF-32 wide character into a multibyte character in the current locale.

Syntax

```
size_t c16rtomb(
    char *mbchar,
    char16_t wchar,
    mbstate_t *state
);
size_t c32rtomb(
    char *mbchar,
    char32_t wchar,
    mbstate_t *state
);
```

Parameters

mbchar
Pointer to an array to store the multibyte converted character.

wchar
A wide character to convert.

state
A pointer to an **mbstate_t** object.

Return Value

The number of bytes stored in array object *mbchar*, including any shift sequences. If *wchar* is not a valid wide character, the value (**size_t**)(-1) is returned, **errno** is set to **EILSEQ**, and the value of *state* is unspecified.

Remarks

The **c16rtomb** function converts the UTF-16 character *wchar* to the equivalent multibyte narrow character sequence in the current locale. If *mbchar* is not a null pointer, the function stores the converted sequence in the array object pointed to by *mbchar*. Up to **MB_CUR_MAX** bytes are stored in *mbchar*, and *state* is set to the resulting multibyte shift state. If *wchar* is a null wide character, a sequence required to restore the initial shift state is stored, if needed, followed by the null character, and *state* is set to the initial conversion state. The **c32rtomb** function is identical, but converts a UTF-32 character.

If *mbchar* is a null pointer, the behavior is equivalent to a call to the function that substitutes an internal buffer for *mbchar* and a wide null character for *wchar*.

The *state* conversion state object allows you to make subsequent calls to this function and other restartable functions that maintain the shift state of the multibyte output characters. Results are undefined when you mix the use of restartable and non-restartable functions, or if a call to **setlocale** is made between restartable function calls.

Requirements

ROUTINE	REQUIRED HEADER
c16rtomb, **c32rtomb**	C, C++: <uchar.h>

For compatibility information, see Compatibility.

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbrtoc16, mbrtoc32
wcrtomb
wcrtomb_s

cabs, cabsf, cabsl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the absolute value of a complex number.

Syntax

```
double cabs(
   _Dcomplex z
);
float cabs(
   _Fcomplex z
);  // C++ only
long double cabs(
   _Lcomplex z
);  // C++ only
float cabsf(
   _Fcomplex z
);
long double cabsl(
   _Lcomplex z
);
```

Parameters

z
A complex number.

Return Value

The absolute value of *z*.

Remarks

Because C++ allows overloading, you can call overloads of **cabs** that take **_Fcomplex** or **_Lcomplex** values, and return **float** or **long double** values. In a C program, **cabs** always takes a **_Dcomplex** value and returns a **double** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cabs, **cabsf**, **cabsl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
norm, normf, norml
creal, crealf, creall
cproj, cprojf, cprojl

conj, conjf, conjl
cimag, cimagf, cimagl
carg, cargf, cargl

_cabs

10/31/2018 • 2 minutes to read • Edit Online

Calculates the absolute value of a complex number.

Syntax

```
double _cabs(
   struct _complex z
);
```

Parameters

z
Complex number.

Return Value

_cabs returns the absolute value of its argument if successful. On overflow, **_cabs** returns **HUGE_VAL** and sets **errno** to **ERANGE**. You can change error handling with _matherr.

Remarks

The **_cabs** function calculates the absolute value of a complex number, which must be a structure of type _complex. The structure *z* is composed of a real component *x* and an imaginary component *y*. A call to **_cabs** produces a value equivalent to that of the expression `sqrt( z.x * z.x + z.y * z.y )`.

Requirements

ROUTINE	REQUIRED HEADER
_cabs	<math.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cabs.c
// Using _cabs, this program calculates
// the absolute value of a complex number.

#include <math.h>
#include <stdio.h>

int main( void )
{
   struct _complex number = { 3.0, 4.0 };
   double d;

   d = _cabs( number );
   printf( "The absolute value of %f + %fi is %f\n",
           number.x, number.y, d );
}
```

```
The absolute value of 3.000000 + 4.000000i is 5.000000
```

See also

Floating-Point Support
abs, labs, llabs, _abs64
fabs, fabsf, fabsl

cacos, cacosf, cacosl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the arccosine of a complex number, with branch cuts outside the interval [-1, +1] along the real axis.

Syntax

```
_Dcomplex cacos( _Dcomplex z );
_Fcomplex cacosf( _Fcomplex z );
_Lcomplex cacosl( _Lcomplex z );
```

```
_Fcomplex cacos( _Fcomplex z );  // C++ only
_Lcomplex cacos( _Lcomplex z );  // C++ only
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The arccosine of *z*, in radians. The result is unbounded along the imaginary axis, and in the in the interval [0, π] along the real axis. A domain error will occur if *z* is outside the interval [-1, +1].

Remarks

Because C++ allows overloading, you can call overloads of **cacos** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **cacos** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cacos, **cacosf**, **cacosl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl
csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
ctan, ctanf, ctanl
csin, csinf, csinl

casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

cacosh, cacoshf, cacoshl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the inverse hyperbolic cosine of a complex number with a branch cut at values less than 1 along the real axis. .

Syntax

```
_Dcomplex cacosh(
   _Dcomplex z
);
_Fcomplex cacosh(
   _Fcomplex z
);  // C++ only
_Lcomplex cacosh(
   _Lcomplex z
);  // C++ only
_Fcomplex cacoshf(
   _Fcomplex z
);
_Lcomplex cacoshl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The inverse hyperbolic cosine of *z*, in radians. The result is unbounded and non-negative along the real axis, and in the interval [-iπ, +iπ] along the imaginary axis.

Remarks

Because C++ allows overloading, you can call overloads of **cacosh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **cacosh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cacosh, **cacoshf**, **cacoshl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl

catan, catanf, catanl
csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

_callnewh

10/31/2018 • 2 minutes to read • Edit Online

Calls the currently installed *new handler*.

Syntax

```
int _callnewh(
   size_t size
   )
```

Parameters

size
The amount of memory that the new operator tried to allocate.

Return Value

VALUE	DESCRIPTION
0	Failure: Either no new handler is installed or no new handler is active.
1	Success: The new handler is installed and active. The memory allocation can be retried.

Exceptions

This function throws bad_alloc if the *new handler* can't be located.

Remarks

The *new handler* is called if the new operator fails to successfully allocate memory. The new handler might then initiate some appropriate action, such as freeing memory so that subsequent allocations succeed.

Requirements

ROUTINE	REQUIRED HEADER
_callnewh	internal.h

See also

_set_new_handler
_set_new_mode

calloc

10/31/2018 • 2 minutes to read • Edit Online

Allocates an array in memory with elements initialized to 0.

Syntax

```
void *calloc(
   size_t num,
   size_t size
);
```

Parameters

number
Number of elements.

size
Length in bytes of each element.

Return Value

calloc returns a pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **calloc** function allocates storage space for an array of *number* elements, each of length *size* bytes. Each element is initialized to 0.

calloc sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

calloc calls **malloc** to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **calloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1);
```

early in your program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **calloc** resolves to _calloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

calloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and

restrict.

Requirements

ROUTINE	REQUIRED HEADER
calloc	<stdlib.h> and <malloc.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_calloc.c
// This program uses calloc to allocate space for
// 40 long integers. It initializes each element to zero.

#include <stdio.h>
#include <malloc.h>

int main( void )
{
   long *buffer;

   buffer = (long *)calloc( 40, sizeof( long ) );
   if( buffer != NULL )
      printf( "Allocated 40 long integers\n" );
   else
      printf( "Can't allocate memory\n" );
   free( buffer );
}
```

```
Allocated 40 long integers
```

See also

Memory Allocation
free
malloc
realloc

_calloc_dbg

10/31/2018 • 2 minutes to read • Edit Online

Allocates a number of memory blocks in the heap with additional space for a debugging header and overwrite buffers (debug version only).

Syntax

```
void *_calloc_dbg(
   size_t num,
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

number
Requested number of memory blocks.

size
Requested size of each memory block (bytes).

blockType
Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

For information about the allocation block types and how they are used, seeTypes of blocks on the debug heap.

filename
Pointer to name of the source file that requested allocation operation or **NULL**.

linenumber
Line number in the source file where allocation operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_calloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function returns a pointer to the user portion of the last allocated memory block, calls the new handler function, or returns **NULL**. For a complete description of the return behavior, see the Remarks section. For more information about how the new handler function is used, see the calloc function.

Remarks

_calloc_dbg is a debug version of the calloc function. When _DEBUG is not defined, each call to **_calloc_dbg** is reduced to a call to **calloc**. Both **calloc** and **_calloc_dbg** allocate *number* memory blocks in the base heap, but **_calloc_dbg** offers several debugging features:

- Buffers on either side of the user portion of the block to test for leaks.
- A block type parameter to track specific allocation types.

- *filename/linenumber* information to determine the origin of allocation requests.

_calloc_dbg allocates each memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

_calloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails; **EINVAL** is returned if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the differences between calling a standard heap function versus its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_calloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Example

```
// crt_callocd.c
// This program uses _calloc_dbg to allocate space for
// 40 long integers. It initializes each element to zero.

#include <stdio.h>
#include <malloc.h>
#include <crtdbg.h>

int main( void )
{
    long *bufferN, *bufferC;

    // Call _calloc_dbg to include the filename and line number
    // of our allocation request in the header and also so we can
    // allocate CLIENT type blocks specifically
    bufferN = (long *)_calloc_dbg( 40, sizeof(long), _NORMAL_BLOCK, __FILE__, __LINE__ );
    bufferC = (long *)_calloc_dbg( 40, sizeof(long), _CLIENT_BLOCK, __FILE__, __LINE__ );
    if( bufferN != NULL && bufferC != NULL )
        printf( "Allocated memory successfully\n" );
    else
        printf( "Problem allocating memory\n" );

    / _free_dbg must be called to free CLIENT type blocks
    free( bufferN );
    _free_dbg( bufferC, _CLIENT_BLOCK );
}
```

```
Allocated memory successfully
```

See also

Debug Routines
calloc
_malloc_dbg
_DEBUG

carg, cargf, cargl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the argument of a complex number, with a branch cut along the negative real axis.

Syntax

```
double carg(
   _Dcomplex z
);
float carg(
   _Fcomplex z
);  // C++ only
long double carg(
   _Lcomplex z
);  // C++ only
float cargf(
   _Fcomplex z
);
long double cargl(
   _Lcomplex z
);
```

Parameters

z
A complex number.

Return Value

The argument (also known as the phase) of *z*. The result is in the interval [-π, +π].

Remarks

Because C++ allows overloading, you can call overloads of **carg** that take **_Fcomplex** or **_Lcomplex** values, and return **float** or **long double** values. In a C program, **carg** always takes a **_Dcomplex** value and returns a **double** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
carg, **cargf**, **cargl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
norm, normf, norml
creal, crealf, creall
cproj, cprojf, cprojl

conj, conjf, conjl
cimag, cimagf, cimagl
cabs, cabsf, cabsl

casin, casinf, casinl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the arcsine of a complex number, with branch cuts outside the interval [-1, +1] along the real axis.

Syntax

```
_Dcomplex casin(
   _Dcomplex z
);
_Fcomplex casin(
   _Fcomplex z
);  // C++ only
_Lcomplex casin(
   _Lcomplex z
);  // C++ only
_Fcomplex casinf(
   _Fcomplex z
);
_Lcomplex casinl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The arcsine of *z*, in radians. The result is unbounded along the imaginary axis, and in the interval [-π/2, +π/2] along the real axis.

Remarks

Because C++ allows overloading, you can call overloads of **casin** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **casin** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
casin, **casinf**, **casinl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl

csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

casinh, casinhf, casinhl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the inverse hyperbolic sine of a complex number, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Syntax

```
_Dcomplex casinh(
   _Dcomplex z
);
_Fcomplex casinh(
   _Fcomplex z
);  // C++ only
_Lcomplex casinh(
   _Lcomplex z
);  // C++ only
_Fcomplex casinhf(
   _Fcomplex z
);
_Lcomplex casinhl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The inverse hyperbolic sine of *z*, in radians. The result is unbound along the real axis, and in the interval [-iπ/2, +iπ/2] along the imaginary axis.

Remarks

Because C++ allows overloading, you can call overloads of **casinh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **casinh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
casinh, **casinhf**, **casinhl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl

catan, catanf, catanl
csinh, csinhf, csinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

catan, catanf, catanl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the arctangent of a complex number with branch cuts outside the interval [-1; +1] along the imaginary axis.

Syntax

```
_Dcomplex catan( _Dcomplex z );
_Fcomplex catanf( _Fcomplex z );
_Lcomplex catanl( _Lcomplex z );
```

```
_Fcomplex catan( _Fcomplex z );  // C++ only
_Lcomplex catan( _Lcomplex z );  // C++ only
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The arctangent of *z*, in radians. The result is unbounded along the imaginary axis, and in the interval [-π/2; +π/2] along the real axis.

Remarks

Because C++ allows overloading, you can call overloads of **catan** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **catan** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
catan, **catanf**, **catanl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl

csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

catanh, catanhf, catanhl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the inverse hyperbolic tangent of a complex number, with branch cuts outside the interval [-1; +1] along the real axis.

Syntax

```
_Dcomplex catanh(
   _Dcomplex z
);
_Fcomplex catanh(
   _Fcomplex z
);  // C++ only
_Lcomplex catanh(
   _Lcomplex z
);  //  C++ only
_Fcomplex catanhf(
   _Fcomplex z
);
_Lcomplex catanhl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The inverse hyperbolic tangent of *z*, in radians. The result is unbounded along the real axis, and in the interval [-iπ/2; +iπ/2] along the imaginary axis. A domain error will occur if *z* is outside the interval [-1, +1]. A pole error will occur if *z* is -1 or +1.

Remarks

Because C++ allows overloading, you can call overloads of **catanh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **catanh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
catanh, **catanhf**, **catanhl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
ctanh, ctanhf, ctanhl

catan, catanf, catanl
csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

cbrt, cbrtf, cbrtl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the cube root.

Syntax

```
double cbrt(
   double x
);
float cbrt(
   float x
);  // C++ only
long double cbrt(
   long double x
);  // C++ only
float cbrtf(
   float x
);
long double cbrtl(
   long double x
);
```

Parameters

x
Floating-point value

Return Value

The **cbrt** functions return the cube-root of *x*.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± ∞, QNAN, IND	none	none

Remarks

Because C++ allows overloading, you can call overloads of **cbrt** that take **float** or **long double** types. In a C program, **cbrt** always takes and returns **double**.

Requirements

FUNCTION	C HEADER	C++ HEADER
cbrt, **cbrtf**, **cbrtl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

Example

```
// crt_cbrt.c
// Compile using: cl /W4 crt_cbrt.c
// This program calculates a cube root.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double question = -64.64;
   double answer;

   answer = cbrt(question);
   printf("The cube root of %.2f is %.6f\n", question, answer);
}
```

```
The cube root of -64.64 is -4.013289
```

See also

Floating-Point Support
exp, expf, expl
log, logf, log10, log10f
pow, powf, powl

_Cbuild, _FCbuild, _LCbuild

10/31/2018 • 2 minutes to read • Edit Online

Constructs a complex number from real and imaginary parts.

Syntax

```
_Dcomplex _Cbuild( double real, double imaginary );
_Fcomplex _FCbuild( float real, float imaginary );
_Lcomplex _LCbuild( long double real, long double imaginary );
```

Parameters

real
The real part of the complex number to construct.

imaginary
The imaginary part of the complex number to construct.

Return Value

A **_Dcomplex**, **_Fcomplex**, or **_Lcomplex** structure that represents the complex number (*real*, *imaginary* * i) for values of the specified floating-point type.

Remarks

The **_Cbuild**, **_FCbuild**, and **_LCbuild** functions simplify creation of complex types. Use the creal, crealf, creall and cimag, cimagf, cimagl functions to retrieve the real and imaginary portions of the represented complex numbers.

Requirements

ROUTINE	C HEADER	C++ HEADER
_Cbuild, **_FCbuild**, **_LCbuild**	<complex.h>	<ccomplex>

These functions are Microsoft-specific. The types **_Dcomplex**, **_Fcomplex**, and **_Lcomplex** are Microsoft-specific equivalents to the unimplemented C99 native types **double _Complex**, **float _Complex**, and **long double _Complex**, respectively. For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
_Cmulcc, _FCmulcc, _LCmulcc
_Cmulcr, _FCmulcr, _LCmulcr
norm, normf, norml
cproj, cprojf, cprojl
conj, conjf, conjl
creal, crealf, creall
cimag, cimagf, cimagl
carg, cargf, cargl

cabs, cabsf, cabsl

ccos, ccosf, ccosl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the cosine of a complex number.

Syntax

```
_Dcomplex ccos(
   _Dcomplex z
);
_Fcomplex ccos(
   _Fcomplex z
);  // C++ only
_Lcomplex ccos(
   _Lcomplex z
);  // C++ only
_Fcomplex ccosf(
   _Fcomplex z
);
_Lcomplex ccosl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents the angle, in radians.

Return Value

The cosine of *z*, in radians.

Remarks

Because C++ allows overloading, you can call overloads of **ccos** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **ccos** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
ccos, **ccosf**, **ccosl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl
csinh, csinhf, csinhl

casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
csqrt, csqrtf, csqrtl

ccosh, ccoshf, ccoshl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the hyperbolic cosine of a complex number.

Syntax

```
_Dcomplex ccosh(
   _Dcomplex z
);
_Fcomplex ccosh(
   _Fcomplex z
);  // C++ only
_Lcomplex ccosh(
   _Lcomplex z
);  // C++ only
_Fcomplex ccoshf(
   _Fcomplex z
);
_Lcomplex ccoshl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents the angle, in radians.

Return Value

The hyperbolic cosine of *z*, in radians.

Remarks

Because C++ allows overloading, you can call overloads of **ccosh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **ccosh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
ccosh, **ccoshf**, **ccoshl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl
csinh, csinhf, csinhl

casinh, casinhf, casinhl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

ceil, ceilf, ceill

10/31/2018 • 2 minutes to read • Edit Online

Calculates the ceiling of a value.

Syntax

```
double ceil(
   double x
);
float ceil(
   float x
);  // C++ only
long double ceil(
   long double x
);  // C++ only
float ceilf(
   float x
);
long double ceill(
   long double x
);
```

Parameters

x
Floating-point value.

Return Value

The **ceil** functions return a floating-point value that represents the smallest integer that is greater than or equal to *x*. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	none	**_DOMAIN**

ceil has an implementation that uses Streaming SIMD Extensions 2 (SSE2). For information and restrictions about using the SSE2 implementation, see _set_SSE2_enable.

Remarks

Because C++ allows overloading, you can call overloads of **ceil** that take **float** or **long double** types. In a C program, **ceil** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER
ceil, **ceilf**, **ceill**	<math.h>

For additional compatibility information, see Compatibility.

Example

See the example for floor.

See also

Floating-Point Support
floor, floorf, floorl
fmod, fmodf
round, roundf, roundl

_cexit, _c_exit

10/31/2018 • 2 minutes to read • Edit Online

Performs cleanup operations and returns without terminating the process.

Syntax

```
void _cexit( void );
void _c_exit( void );
```

Remarks

The **_cexit** function calls, in last-in, first-out (LIFO) order, the functions registered by **atexit** and **_onexit**. Then **_cexit** flushes all I/O buffers and closes all open streams before returning. **_c_exit** is the same as **_exit** but returns to the calling process without processing **atexit** or **_onexit** or flushing stream buffers. The behavior of **exit**, **_exit**, **_cexit**, and **_c_exit** is shown in the following table.

FUNCTION	BEHAVIOR
exit	Performs complete C library termination procedures, terminates process, and exits with supplied status code.
_exit	Performs quick C library termination procedures, terminates process, and exits with supplied status code.
_cexit	Performs complete C library termination procedures and returns to caller, but does not terminate process.
_c_exit	Performs quick C library termination procedures and returns to caller, but does not terminate process.

When you call the **_cexit** or **_c_exit** functions, the destructors for any temporary or automatic objects that exist at the time of the call are not called. An automatic object is an object that is defined in a function where the object is not declared to be static. A temporary object is an object created by the compiler. To destroy an automatic object before calling **_cexit** or **_c_exit**, explicitly call the destructor for the object, as follows:

```
myObject.myClass::~myClass( );
```

Requirements

ROUTINE	REQUIRED HEADER
_cexit	<process.h>
_c_exit	<process.h>

For more compatibility information, see Compatibility.

See also

Process and Environment Control
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

cexp, cexpf, cexpl

10/31/2018 • 2 minutes to read • Edit Online

Compute the base-e exponential of a complex number.

Syntax

```
_Dcomplex cexp( _Dcomplex z );
_Fcomplex cexpf( _Fcomplex z );
_Lcomplex cexpl( _Lcomplex z );
```

```
_Fcomplex cexp( _Fcomplex z );  // C++ only
_Lcomplex cexp( _Lcomplex z );  // C++ only
```

Parameters

z
A complex number that represents the exponent.

Return Value

The value of **e** raised to the power of *z*.

Remarks

Because C++ allows overloading, you can call overloads of **cexp** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **cexp** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cexp, **cexpf**, **cexpl**	<complex.h>	<complex.h>

For compatibility information, see Compatibility.

See also

Alphabetical Function Reference
cpow, cpowf, cpowl
clog10, clog10f, clog10l
clog, clogf, clogl

cgets

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _cgets or security-enhanced _cgets_s instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_cgets_s, _cgetws_s

10/31/2018 • 2 minutes to read • Edit Online

Gets a character string from the console. These versions of _cgets and _cgetws have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _cgets_s(
   char *buffer,
   size_t numberOfElements,
   size_t *pSizeRead
);
errno_t _cgetws_s(
   wchar_t *buffer
   size_t numberOfElements,
   size_t *pSizeRead
);
template <size_t size>
errno_t _cgets_s(
   char (&buffer)[size],
   size_t *pSizeRead
); // C++ only
template <size_t size>
errno_t _cgetws_s(
   wchar_t (&buffer)[size],
   size_t *pSizeRead
); // C++ only
```

Parameters

buffer
Storage location for data.

numberOfElements
The size of the buffer in single-byte or wide characters, which is also the maximum number of characters to be read.

pSizeRead
The number of characters actually read.

Return Value

The return value is zero if successful; otherwise, an error code if a failure occurs.

Error Conditions

BUFFER	*NUMBEROFELEMENTS*	*PSIZEREAD*	RETURN	CONTENTS OF *BUFFER*
NULL	any	any	**EINVAL**	n/a
not **NULL**	zero	any	**EINVAL**	not modified
not **NULL**	any	**NULL**	**EINVAL**	zero-length string

Remarks

_cgets_s and **_cgetws_s** read a string from the console and copy the string (with a null terminator) into *buffer*. **_cgetws_s** is the wide character version of the function; other than the character size, the behavior of these two functions is identical. The maximum size of the string to be read is passed in as the *numberOfElements* parameter. This size should include an extra character for the terminating null. The actual number of characters read is placed in *pSizeRead*.

If an error occurs during the operation or in the validating of the parameters, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **errno** is set to **EINVAL** and **EINVAL** is returned.

In C++, the use of these functions is simplified by template overloads; the overloads can infer buffer length automatically, thereby eliminating the need to specify a size argument, and they can automatically replace older, less-secure functions with their newer, more secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_cgetts_s	**_cgets_s**	**_cgets_s**	**_cgetws_s**

Requirements

ROUTINE	REQUIRED HEADER
_cgets_s	<conio.h>
_cgetws_s	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

See also

Console and Port I/O
_getch, _getwch

chdir

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _chdir instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_chdir, _wchdir

10/31/2018 • 2 minutes to read • Edit Online

Changes the current working directory.

Syntax

```
int _chdir(
   const char *dirname
);
int _wchdir(
   const wchar_t *dirname
);
```

Parameters

dirname
Path of new working directory.

Return Value

These functions return a value of 0 if successful. A return value of -1 indicates failure. If the specified path could not be found, **errno** is set to **ENOENT**. If *dirname* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Remarks

The **_chdir** function changes the current working directory to the directory specified by *dirname*. The *dirname* parameter must refer to an existing directory. This function can change the current working directory on any drive. If a new drive letter is specified in *dirname*, the default drive letter is changed as well. For example, if A is the default drive letter and \BIN is the current working directory, the following call changes the current working directory for drive C and establishes C as the new default drive:

```
_chdir("c:\temp");
```

When you use the optional backslash character (****) in paths, you must place two backslashes (****) in a C string literal to represent a single backslash (****).

_wchdir is a wide-character version of **_chdir**; the *dirname* argument to **_wchdir** is a wide-character string. **_wchdir** and **_chdir** behave identically otherwise.

Generic-Text Routine Mapping:

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tchdir	**_chdir**	**_chdir**	**_wchdir**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_chdir	<direct.h>	<errno.h>
_wchdir	<direct.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_chdir.c
// arguments: C:\WINDOWS

/* This program uses the _chdir function to verify
   that a given directory exists. */

#include <direct.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( int argc, char *argv[] )
{

   if(_chdir( argv[1] ) )
   {
      switch (errno)
      {
      case ENOENT:
         printf( "Unable to locate the directory: %s\n", argv[1] );
         break;
      case EINVAL:
         printf( "Invalid buffer.\n");
         break;
      default:
         printf( "Unknown error.\n");
      }
   }
   else
      system( "dir *.exe");
}
```

```
Volume in drive C has no label.
Volume Serial Number is 2018-08A1

Directory of c:\windows

08/29/2002  04:00 AM         1,004,032 explorer.exe
12/17/2002  04:43 PM            10,752 hh.exe
03/03/2003  09:24 AM            33,792 ieuninst.exe
10/29/1998  04:45 PM           306,688 IsUninst.exe
08/29/2002  04:00 AM            66,048 NOTEPAD.EXE
03/03/2003  09:24 AM            33,792 Q330994.exe
08/29/2002  04:00 AM           134,144 regedit.exe
02/28/2003  06:26 PM            46,352 setdebug.exe
08/29/2002  04:00 AM            15,360 TASKMAN.EXE
08/29/2002  04:00 AM            49,680 twunk_16.exe
08/29/2002  04:00 AM            25,600 twunk_32.exe
08/29/2002  04:00 AM           256,192 winhelp.exe
08/29/2002  04:00 AM           266,752 winhlp32.exe
              13 File(s)      2,249,184 bytes
               0 Dir(s)  67,326,029,824 bytes free
```

See also

Directory Control
_mkdir, _wmkdir
_rmdir, _wrmdir
system, _wsystem

_chdrive

10/31/2018 • 2 minutes to read • Edit Online

Changes the current working drive.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _chdrive(
   int drive
);
```

Parameters

drive
An integer from 1 through 26 that specifies the current working drive (1=A, 2=B, and so forth).

Return Value

Zero (0) if the current working drive was changed successfully; otherwise, -1.

Remarks

If *drive* is not in the range from 1 through 26, the invalid-parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, the **_chdrive** function returns -1, **errno** is set to **EACCES**, and **_doserrno** is set to **ERROR_INVALID_DRIVE**.

The **_chdrive** function is not thread-safe because it depends on the **SetCurrentDirectory** function, which is itself not thread-safe. To use **_chdrive** safely in a multi-threaded application, you must provide your own thread synchronization. For more information, see SetCurrentDirectory.

The **_chdrive** function changes only the current working drive; **_chdir** changes the current working directory.

Requirements

ROUTINE	REQUIRED HEADER
_chdrive	<direct.h>

For more information, see Compatibility.

Example

See the example for _getdrive.

See also

Directory Control
_chdir, _wchdir
_fullpath, _wfullpath
_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir
system, _wsystem

_chgsign, _chgsignf, _chgsignl

10/31/2018 • 2 minutes to read • Edit Online

Reverses the sign of a floating-point argument.

Syntax

```
double _chgsign(
   double x
);
float _chgsignf(
   float x
);
long double _chgsignl(
   long double x
);
```

Parameters

x
The floating-point value to be changed.

Return Value

The **_chgsign** functions return a value that's equal to the floating-point argument *x*, but with its sign reversed. There is no error return.

Requirements

ROUTINE	REQUIRED HEADER
_chgsign	<float.h>
_chgsignf, **_chgsignl**	<math.h>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
fabs, fabsf, fabsl
copysign, copysignf, copysignl, _copysign, _copysignf, _copysignl

chmod

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _chmod instead.

_chmod, _wchmod

11/9/2018 • 2 minutes to read • Edit Online

Changes the file-permission settings.

Syntax

```
int _chmod( const char *filename, int pmode );
int _wchmod( const wchar_t *filename, int pmode );
```

Parameters

filename
Name of the existing file.

pmode
Permission setting for the file.

Return Value

These functions return 0 if the permission setting is successfully changed. A return value of -1 indicates failure. If the specified file could not be found, **errno** is set to **ENOENT**; if a parameter is invalid, **errno** is set to **EINVAL**.

Remarks

The **_chmod** function changes the permission setting of the file specified by *filename*. The permission setting controls the read and write access to the file. The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\Stat.h.

PMODE	MEANING
_S_IREAD	Only reading permitted.
_S_IWRITE	Writing permitted. (In effect, permits reading and writing.)
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

When both constants are given, they are joined with the bitwise or operator (**|**). If write permission is not given, the file is read-only. Note that all files are always readable; it is not possible to give write-only permission. Thus, the modes **_S_IWRITE** and **_S_IREAD** | **_S_IWRITE** are equivalent.

_wchmod is a wide-character version of **_chmod**; the *filename* argument to **_wchmod** is a wide-character string. **_wchmod** and **_chmod** behave identically otherwise.

This function validates its parameters. If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function simply ignores those. If *filename* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tchmod	**_chmod**	**_chmod**	**_wchmod**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_chmod	<io.h>	<sys/types.h>, <sys/stat.h>, <errno.h>
_wchmod	<io.h> or <wchar.h>	<sys/types.h>, <sys/stat.h>, <errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_chmod.c
// This program uses _chmod to
// change the mode of a file to read-only.
// It then attempts to modify the file.
//

#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

// Change the mode and report error or success
void set_mode_and_report(char * filename, int mask)
{
   // Check for failure
   if( _chmod( filename, mask ) == -1 )
   {
      // Determine cause of failure and report.
      switch (errno)
      {
         case EINVAL:
            fprintf( stderr, "Invalid parameter to chmod.\n");
            break;
         case ENOENT:
            fprintf( stderr, "File %s not found\n", filename );
            break;
         default:
            // Should never be reached
            fprintf( stderr, "Unexpected error in chmod.\n" );
       }
   }
   else
   {
      if (mask == _S_IREAD)
        printf( "Mode set to read-only\n" );
      else if (mask & _S_IWRITE)
        printf( "Mode set to read/write\n" );
   }
   fflush(stderr);
}

int main( void )
{

   // Create or append to a file.
   system( "echo /* End of file */ >> crt_chmod.c_input" );

   // Set file mode to read-only:
   set_mode_and_report("crt_chmod.c_input ", _S_IREAD );

   system( "echo /* End of file */ >> crt_chmod.c_input " );

   // Change back to read/write:
   set_mode_and_report("crt_chmod.c_input ", _S_IWRITE );

   system( "echo /* End of file */ >> crt_chmod.c_input " );
}
```

```
A line of text.
```

```
      A line of text.Mode set to read-only
Access is denied.
Mode set to read/write
```

See also

File Handling
_access, _waccess
_creat, _wcreat
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_open, _wopen
_stat, _wstat Functions

chsize

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _chsize or security-enhanced _chsize_s instead.

_chsize

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a file. A more secure version is available; see _chsize_s.

Syntax

```
int _chsize(
   int fd,
   long size
);
```

Parameters

fd
File descriptor referring to an open file.

size
New length of the file in bytes.

Return Value

_chsize returns the value 0 if the file size is successfully changed. A return value of -1 indicates an error: **errno** is set to **EACCES** if the specified file is read-only or the specified file is locked against access, to **EBADF** if the descriptor is invalid, **ENOSPC** if no space is left on the device, or **EINVAL** if *size* is less than zero.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_chsize** function extends or truncates the file associated with *fd* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

This function validates its parameters. If *size* is less than zero or *fd* is a bad file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_chsize	<io.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_chsize.c
// This program uses _filelength to report the size
// of a file before and after modifying it with _chsize.

#include <io.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <share.h>

int main( void )
{
   int fh, result;
   unsigned int nbytes = BUFSIZ;

   // Open a file
   if( _sopen_s( &fh, "data", _O_RDWR | _O_CREAT, _SH_DENYNO,
                 _S_IREAD | _S_IWRITE ) == 0 )
   {
      printf( "File length before: %ld\n", _filelength( fh ) );
      if( ( result = _chsize( fh, 329678 ) ) == 0 )
         printf( "Size successfully changed\n" );
      else
         printf( "Problem in changing the size\n" );
      printf( "File length after:  %ld\n", _filelength( fh ) );
      _close( fh );
   }
}
```

```
File length before: 0
Size successfully changed
File length after:  329678
```

See also

File Handling
_close
_sopen, _wsopen
_open, _wopen

_chsize_s

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a file. This is a version of _chsize with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _chsize_s(
   int fd,
   __int64 size
);
```

Parameters

fd
File descriptor referring to an open file.

size
New length of the file in bytes.

Return Value

_chsize_s returns the value 0 if the file size is successfully changed. A nonzero return value indicates an error: the return value is **EACCES** if the specified file is locked against access, **EBADF** if the specified file is read-only or the descriptor is invalid, **ENOSPC** if no space is left on the device, or **EINVAL** if size is less than zero. **errno** is set to the same value.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_chsize_s** function extends or truncates the file associated with *fd* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

_chsize_s takes a 64-bit integer as the file size, and therefore can handle file sizes greater than 4 GB. **_chsize** is limited to 32-bit file sizes.

This function validates its parameters. If *fd* is not a valid file descriptor or size is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_chsize_s	<io.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

File Handling
_chsize
_close
_creat, _wcreat
_open, _wopen

cimag, cimagf, cimagl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the imaginary part of a complex number.

Syntax

```
double cimag( _Dcomplex z );
float cimagf( _Fcomplex z );
long double cimagl( _Lcomplex z );
```

```
float cimag( _Fcomplex z );  // C++
long double cimag( _Lcomplex z );  // C++
```

Parameters

z
A complex number.

Return Value

The imaginary part of *z*.

Remarks

Because C++ allows overloading, you can call overloads of **cimag** that take **_Fcomplex** or **_Lcomplex** values, and return **float** or **long double** values. In a C program, **cimag** always takes a **_Dcomplex** value and returns a **double** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cimag, **cimagf**, **cimagl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
norm, normf, norml
creal, crealf, creall
cproj, cprojf, cprojl
conj, conjf, conjl
carg, cargf, cargl
cabs, cabsf, cabsl

_clear87, _clearfp

10/31/2018 • 2 minutes to read • Edit Online

Gets and clears the floating-point status word.

Syntax

```
unsigned int _clear87( void );
unsigned int _clearfp( void );
```

Return Value

The bits in the value returned indicate the floating-point status before the call to **_clear87** or **_clearfp**. For a complete definition of the bits returned by **_clear87**, see Float.h. Many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from **_clear87** and **_status87** become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

Remarks

The **_clear87** function clears the exception flags in the floating-point status word, sets the busy bit to 0, and returns the status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

_clearfp is a platform-independent, portable version of the **_clear87** routine. It is identical to **_clear87** on Intel (x86) platforms and is also supported by the x64 and ARM platforms. To ensure that your floating-point code is portable to x64 and ARM, use **_clearfp**. If you are only targeting x86 platforms, you can use either **_clear87** or **_clearfp**.

These functions are deprecated when compiling with /clr (Common Language Runtime Compilation) because the common language runtime only supports the default floating-point precision.

Requirements

ROUTINE	REQUIRED HEADER
_clear87	<float.h>
_clearfp	<float.h>

For more compatibility information, see Compatibility.

Example

```
// crt_clear87.c
// compile with: /Od

// This program creates various floating-point
// problems, then uses _clear87 to report on these problems.
// Compile this program with Optimizations disabled (/Od).
// Otherwise the optimizer will remove the code associated with
// the unused floating-point values.
//

#include <stdio.h>
#include <float.h>

int main( void )
{
   double a = 1e-40, b;
   float x, y;

   printf( "Status: %.4x - clear\n", _clear87()  );

   // Store into y is inexact and underflows:
   y = a;
   printf( "Status: %.4x - inexact, underflow\n", _clear87() );

   // y is denormal:
   b = y;
   printf( "Status: %.4x - denormal\n", _clear87() );
}
```

```
Status: 0000 - clear
Status: 0003 - inexact, underflow
Status: 80000 - denormal
```

See also

Floating-Point Support
_control87, _controlfp, __control87_2
_status87, _statusfp, _statusfp2

clearerr

11/8/2018 • 2 minutes to read • Edit Online

Resets the error indicator for a stream. A more secure version of this function is available; see clearerr_s.

Syntax

```
void clearerr(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Remarks

The **clearerr** function resets the error indicator and end-of-file indicator for *stream*. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr**, fseek, **fsetpos**, or rewind is called.

If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns. For more information on **errno** and error codes, see errno Constants.

A more secure version of this function is available; see clearerr_s.

Requirements

ROUTINE	REQUIRED HEADER
clearerr	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_clearerr.c
// This program creates an error
// on the standard input stream, then clears
// it so that future reads won't fail.

#include <stdio.h>

int main( void )
{
   int c;
   // Create an error by writing to standard input.
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      clearerr( stdin );
   }

   // See if read causes an error.
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      clearerr( stdin );
   }
   else
      printf( "No read error\n" );
}
```

Input

```
n
```

Output

```
Write error: No error
Will input cause an error? n
No read error
```

See also

Error Handling
Stream I/O
_eof
feof
ferror
perror, _wperror

clearerr_s

11/8/2018 • 2 minutes to read • Edit Online

Resets the error indicator for a stream. This is a version of clearerr with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t clearerr_s(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure

Return Value

Zero if successful; **EINVAL** if *stream* is **NULL**.

Remarks

The **clearerr_s** function resets the error indicator and end-of-file indicator for *stream*. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr_s**, **clearerr**, fseek, **fsetpos**, or rewind is called.

If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
clearerr_s	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_clearerr_s.c
// This program creates an error
// on the standard input stream, then clears
// it so that future reads won't fail.

#include <stdio.h>

int main( void )
{
   int c;
   errno_t err;

   // Create an error by writing to standard input.
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      err = clearerr_s( stdin );
      if (err != 0)
      {
         abort();
      }
   }

   // See if read causes an error.
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      err = clearerr_s( stdin );
      if (err != 0)
      {
         abort();
      }
   }
}
```

Input

```
n
```

Output

```
Write error: Bad file descriptor
Will input cause an error? n
```

See also

Error Handling
Stream I/O
clearerr
_eof
feof
ferror
perror, _wperror

clock

10/31/2018 • 2 minutes to read • Edit Online

Calculates the wall-clock time used by the calling process.

Syntax

```
clock_t clock( void );
```

Return Value

The elapsed time since the CRT initialization at the start of the process, measured in **CLOCKS_PER_SEC** units per second. If the elapsed time is unavailable or has exceeded the maximum positive time that can be recorded as a **clock_t** type, the function returns the value `(clock_t)(-1)`.

Remarks

The **clock** function tells how much wall-clock time has passed since the CRT initialization during process start. Note that this function does not strictly conform to ISO C, which specifies net CPU time as the return value. To obtain CPU times, use the Win32 GetProcessTimes function. To determine the elapsed time in seconds, divide the value returned by the **clock** function by the macro **CLOCKS_PER_SEC**.

Given enough time, the value returned by **clock** can exceed the maximum positive value of **clock_t**. When the process has run longer, the value returned by **clock** is always `(clock_t)(-1)`, as specified by the ISO C99 standard (7.23.2.1) and ISO C11 standard (7.27.2.1). Microsoft implements **clock_t** as a **long**, a signed 32-bit integer, and the **CLOCKS_PER_SEC** macro is defined as 1000. This gives a maximum **clock** function return value of 2147483.647 seconds, or about 24.8 days. Do not rely on the value returned by **clock** in processes that have run for longer than this amount of time. You can use the 64-bit time function or the Windows QueryPerformanceCounter function to record process elapsed times of many years.

Requirements

ROUTINE	REQUIRED HEADER
clock	<time.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_clock.c
// This sample uses clock() to 'sleep' for three
// seconds, then determines how long it takes
// to execute an empty loop 600000000 times.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Pauses for a specified number of milliseconds.
void do_sleep( clock_t wait )
{
   clock_t goal;
   goal = wait + clock();
   while( goal > clock() )
      ;
}

const long num_loops = 600000000L;

int main( void )
{
   long    i = num_loops;
   clock_t start, finish;
   double  duration;

   // Delay for a specified time.
   printf( "Delay for three seconds\n" );
   do_sleep( (clock_t)3 * CLOCKS_PER_SEC );
   printf( "Done!\n" );

   // Measure the duration of an event.
   start = clock();
   while( i-- )
      ;
   finish = clock();
   duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf( "Time to do %ld empty loops is ", num_loops );
   printf( "%2.3f seconds\n", duration );
}
```

```
Delay for three seconds
Done!
Time to do 600000000 empty loops is 1.354 seconds
```

See also

Time Management
difftime, _difftime32, _difftime64
time, _time32, _time64

clog, clogf, clogl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the natural logarithm of a complex number, with a branch cut along the negative real axis.

Syntax

```
_Dcomplex clog(
   _Dcomplex z
);
_Fcomplex clog(
   _Fcomplex z
);  // C++ only
_Lcomplex clog(
   _Lcomplex z
);  // C++ only
_Fcomplex clogf(
   _Fcomplex z
);
_Lcomplex clogl(
   _Lcomplex z
);
```

Parameters

z
The base of the logarithm.

Return Value

The natural logarithm of *z*. The result is unbounded along the real axis and in the interval [-iπ, +iπ] along the imaginary axis.

The possible return values are:

Z PARAMETER	RETURN VALUE
Positive	The base 10 logarithm of z
Zero	- ∞
Negative	NaN
NaN	NaN
+ ∞	+ ∞

Remarks

Because C++ allows overloading, you can call overloads of **clog** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **clog** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
clog, **clogf**, **clogl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
cexp, cexpf, cexpl
cpow, cpowf, cpowl
clog10, clog10f, clog10l

clog10, clog10f, clog10l

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the base 10 logarithm of a complex number.

Syntax

```
_Dcomplex clog10( _Dcomplex z );
_Fcomplex clog10f( _Fcomplex z );
_Lcomplex clog10l( _Lcomplex z );
```

```
_Fcomplex clog10( _Fcomplex z );  // C++ only
_Lcomplex clog10( _Lcomplex z );  // C++ only
```

Parameters

z
The base of the logarithm.

Return Value

The possible return values are:

Z PARAMETER	RETURN VALUE
Positive	The base 10 logarithm of z
Zero	- ∞
Negative	NaN
NaN	NaN
+ ∞	+ ∞

Remarks

Because C++ allows overloading, you can call overloads of **clog10** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **clog10** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
clog10, **clog10f**, **clogl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
cexp, cexpf, cexpl
cpow, cpowf, cpowl
clog, clogf, clogl

_close

10/31/2018 • 2 minutes to read • Edit Online

Closes a file.

Syntax

```
int _close(
   int fd
);
```

Parameters

fd
File descriptor referring to the open file.

Return Value

_close returns 0 if the file was successfully closed. A return value of -1 indicates an error.

Remarks

The **_close** function closes the file associated with *fd*.

The file descriptor and the underlying OS file handle are closed. Thus, it is not necessary to call **CloseHandle** if the file was originally opened using the Win32 function **CreateFile** and converted to a file descriptor using **_open_osfhandle**.

This function validates its parameters. If *fd* is a bad file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions returns -1 and **errno** is set to **EBADF**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_close	<io.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example for _open.

See also

Low-Level I/O
_chsize
_creat, _wcreat
_dup, _dup2
_open, _wopen

_unlink, _wunlink

close

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _close instead.

_Cmulcc, _FCmulcc, _LCmulcc

10/31/2018 • 2 minutes to read • Edit Online

Multiplies two complex numbers.

Syntax

```
_Dcomplex _Cmulcc( _Dcomplex x, _Dcomplex y );
_Fcomplex _FCmulcc( _Fcomplex x, _Fcomplex y );
_Lcomplex _LCmulcc( _Lcomplex x, _Lcomplex y );
```

Parameters

x
One of the complex operands to multiply.

y
The other complex operand to multiply.

Return Value

A **_Dcomplex**, **_Fcomplex**, or **_Lcomplex** structure that represents the complex product of the complex numbers *x* and *y*.

Remarks

Because the built-in arithmetic operators do not work on the Microsoft implementation of the complex types, the **_Cmulcc**, **_FCmulcc**, and **_LCmulcc** functions simplify multiplication of complex types.

Requirements

ROUTINE	C HEADER	C++ HEADER
_Cmulcc, **_FCmulcc**, **_LCmulcc**	<complex.h>	<complex.h>

These functions are Microsoft-specific. The types **_Dcomplex**, **_Fcomplex**, and **_Lcomplex** are Microsoft-specific equivalents to the unimplemented C99 native types **double _Complex**, **float _Complex**, and **long double _Complex**, respectively. For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
_Cbuild, _FCbuild, _LCbuild
_Cmulcr, _FCmulcr, _LCmulcr
norm, normf, norml
cproj, cprojf, cprojl
conj, conjf, conjl
creal, crealf, creall
cimag, cimagf, cimagl
carg, cargf, cargl

cabs, cabsf, cabsl

_Cmulcr, _FCmulcr, _LCmulcr

10/31/2018 • 2 minutes to read • Edit Online

Multiplies a complex number by a floating-point number.

Syntax

```
_Dcomplex _Cmulcr( _Dcomplex x, double y );
_Fcomplex _FCmulcr( _Fcomplex x, float y );
_Lcomplex _LCmulcr( _Lcomplex x, long double y );
```

Parameters

x
One of the complex operands to multiply.

y
The floating-point operand to multiply.

Return Value

A **_Dcomplex**, **_Fcomplex**, or **_Lcomplex** structure that represents the complex product of the complex number *x* and flaoting-point number *y*.

Remarks

Because the built-in arithmetic operators do not work on the Microsoft implementation of the complex types, the **_Cmulcr**, **_FCmulcr**, and **_LCmulcr** functions simplify multiplication of complex types by floating-point types.

Requirements

ROUTINE	C HEADER	C++ HEADER
_Cmulcr, **_FCmulcr**, **_LCmulcr**	<complex.h>	<complex.h>

These functions are Microsoft-specific. The types **_Dcomplex**, **_Fcomplex**, and **_Lcomplex** are Microsoft-specific equivalents to the unimplemented C99 native types **double _Complex**, **float _Complex**, and **long double _Complex**, respectively. For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
_Cbuild, _FCbuild, _LCbuild
_Cmulcc, _FCmulcc, _LCmulcc
norm, normf, norml
cproj, cprojf, cprojl
conj, conjf, conjl
creal, crealf, creall
cimag, cimagf, cimagl
carg, cargf, cargl

cabs, cabsf, cabsl

_commit

10/31/2018 • 2 minutes to read • Edit Online

Flushes a file directly to disk.

Syntax

```
int _commit(
   int fd
);
```

Parameters

fd
File descriptor referring to the open file.

Return Value

_commit returns 0 if the file was successfully flushed to disk. A return value of -1 indicates an error.

Remarks

The **_commit** function forces the operating system to write the file associated with *fd* to disk. This call ensures that the specified file is flushed immediately, not at the operating system's discretion.

If *fd* is an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to **EBADF**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_commit	<io.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

Low-Level I/O
_creat, _wcreat
_open, _wopen
_read
_write

compl

11/8/2018 • 2 minutes to read • Edit Online

An alternative to the ~ operator.

Syntax

```
#define compl ~
```

Remarks

The macro yields the operator ~.

Example

```
// iso646_compl.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 1, result;

   result = ~a;
   cout << result << endl;

   result= compl(a);
   cout << result << endl;
}
```

```
-2
-2
```

Requirements

Header: <iso646.h>

_configthreadlocale

11/8/2018 • 2 minutes to read • Edit Online

Configures per-thread locale options.

Syntax

```
int _configthreadlocale( int per_thread_locale_type );
```

Parameters

per_thread_locale_type
The option to set. One of the options listed in the following table.

Return Value

The previous per-thread locale status (**_DISABLE_PER_THREAD_LOCALE** or **_ENABLE_PER_THREAD_LOCALE**), or -1 on failure.

Remarks

The **_configurethreadlocale** function is used to control the use of thread-specific locales. Use one of these *per_thread_locale_type* options to specify or determine the per-thread locale status:

OPTION	DESCRIPTION
_ENABLE_PER_THREAD_LOCALE	Make the current thread use a thread-specific locale. Subsequent calls to **setlocale** in this thread affect only the thread's own locale.
_DISABLE_PER_THREAD_LOCALE	Make the current thread use the global locale. Subsequent calls to **setlocale** in this thread affect other threads using the global locale.
0	Retrieves the current setting for this particular thread.

These functions affect the behavior of **setlocale**, **_tsetlocale**, **_wsetlocale**, and **_setmbcp**. When per-thread locale is disabled, any subsequent call to **setlocale** or **_wsetlocale** changes the locale of all threads that use the global locale. When per-thread locale is enabled, **setlocale** or **_wsetlocale** only affects the current thread's locale.

If you use **_configurethreadlocale** to enable a per-thread locale, we recommend that you call **setlocale** or **_wsetlocale** to set the preferred locale in that thread immediately afterward.

If *per_thread_locale_type* is not one of the values listed in the table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1.

Requirements

ROUTINE	REQUIRED HEADER
_configthreadlocale	<locale.h>

Example

```
// crt_configthreadlocale.cpp
//
// This program demonstrates the use of _configthreadlocale when
// using two independent threads.
//
// Compile by using: cl /EHsc /W4 crt_configthreadlocale.cpp

#include <locale.h>
#include <mbctype.h>
#include <process.h>
#include <windows.h>
#include <stdio.h>
#include <time.h>

#define BUFF_SIZE 100

// Retrieve the date and time in the current
// locale's format.
int get_time(unsigned char* str)
{
    __time64_t  ltime;
    struct tm   thetime;

    // Retieve the time
    _time64(&ltime);
    _gmtime64_s(&thetime, &ltime);

    // Format the current time structure into a string
    // using %#x is the long date representation,
    // appropriate to the current locale
    if (!strftime((char *)str, BUFF_SIZE, "%#x",
                  (const struct tm*)&thetime))
    {
        printf("strftime failed!\n");
        return -1;
    }
    return 0;
}

// This thread sets its locale to German
// and prints the time.
unsigned __stdcall SecondThreadFunc( void* /*pArguments*/ )
{
    unsigned char str[BUFF_SIZE];

    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);

    // Set the thread code page
    _setmbcp(_MB_CP_ANSI);

    // Set the thread locale
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, "German"));

    // Retrieve the time string from the helper function
    if (get_time(str) == 0)
    {
        printf("The time in German locale is: '%s'\n", str);
    }
```

```
    _endthreadex( 0 );
    return 0;
}

// The main thread spawns a second thread (above) and then
// sets the locale to English and prints the time.
int main()
{
    HANDLE          hThread;
    unsigned        threadID;
    unsigned char   str[BUFF_SIZE];

    // Enable per-thread locale causes all subsequent locale
    // setting changes in this thread to only affect this thread.
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);

    // Retrieve the time string from the helper function
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, "English"));

    // Create the second thread.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc,
                                      NULL, 0, &threadID );

    if (get_time(str) == 0)
    {
        // Retrieve the time string from the helper function
        printf("The time in English locale is: '%s'\n\n", str);
    }

    // Wait for the created thread to finish.
    WaitForSingleObject( hThread, INFINITE );

    // Destroy the thread object.
    CloseHandle( hThread );
}
```

```
The thread locale is now set to English_United States.1252.
The time in English locale is: 'Wednesday, May 12, 2004'

The thread locale is now set to German_Germany.1252.
The time in German locale is: 'Mittwoch, 12. Mai 2004'
```

See also

setlocale, _wsetlocale
_beginthread, _beginthreadex
Locale
Multithreading and Locales

conj, conjf, conjl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the complex conjugate of a complex number.

Syntax

```
_Dcomplex conj(
   _Dcomplex z
);
_Fcomplex conj(
   _Fcomplex z
);  // C++ only
_Lcomplex conj(
   _Lcomplex z
);  // C++ only
_Fcomplex conjf(
   _Fcomplex z
);
_Lcomplex conjl(
   _Lcomplex z
);
```

Parameters

z
A complex number.

Return Value

The complex conjugate of *z*. The result has the same real and imaginary part as *z*, but with the opposite sign.

Remarks

Because C++ allows overloading, you can call overloads of **conj** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **conj** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
conj, **conjf**, **conjl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
norm, normf, norml
creal, crealf, creall
cproj, cprojf, cprojl
cimag, cimagf, cimagl

carg, cargf, cargl
cabs, cabsf, cabsl

_control87, _controlfp, __control87_2

10/31/2018 • 5 minutes to read • Edit Online

Gets and sets the floating-point control word. A more secure version of **_controlfp** is available; see _controlfp_s.

Syntax

```
unsigned int _control87(
   unsigned int new,
   unsigned int mask
);
unsigned int _controlfp(
   unsigned int new,
   unsigned int mask
);
int __control87_2(
   unsigned int new,
   unsigned int mask,
   unsigned int* x86_cw,
   unsigned int* sse2_cw
);
```

Parameters

new
New control-word bit values.

mask
Mask for new control-word bits to set.

x86_cw
Filled in with the control word for the x87 floating-point unit. Pass in 0 (**NULL**) to set only the SSE2 control word.

sse2_cw
Control word for the SSE floating-point unit. Pass in 0 (**NULL**) to set only the x87 control word.

Return Value

For **_control87** and **_controlfp**, the bits in the value returned indicate the floating-point control state. For a complete definition of the bits that are returned by **_control87**, see FLOAT.H.

For **__control87_2**, the return value is 1, which indicates success.

Remarks

The **_control87** function gets and sets the floating-point control word. The floating-point control word enables the program to change the precision, rounding, and infinity modes in the floating-point math package, depending on the platform. You can also use **_control87** to mask or unmask floating-point exceptions. If the value for *mask* is equal to 0, **_control87** gets the floating-point control word. If *mask* is nonzero, a new value for the control word is set: For any bit that is on (that is, equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. In other words, **fpcntrl** = ((**fpcntrl** & ~*mask*) | (*new* & *mask*)) where **fpcntrl** is the floating-point control word.

> **NOTE**
>
> By default, the run-time libraries mask all floating-point exceptions.

_controlfp is a platform-independent, portable version of **_control87**. It is nearly identical to the **_control87** function on x86, x64, and ARM platforms. If you are targeting x86, x64, or ARM platforms, use **_control87** or **_controlfp**.

The difference between **_control87** and **_controlfp** is in how they treat DENORMAL values. For x86, x64, and ARM platforms, **_control87** can set and clear the DENORMAL OPERAND exception mask. **_controlfp** does not modify the DENORMAL OPERAND exception mask. This example demonstrates the difference:

```
_control87( _EM_INVALID, _MCW_EM );
// DENORMAL is unmasked by this call
_controlfp( _EM_INVALID, _MCW_EM );
// DENORMAL exception mask remains unchanged
```

The possible values for the mask constant (*mask*) and new control values (*new*) are shown in the following Hexadecimal Values table. Use the portable constants listed below (**_MCW_EM**, **_EM_INVALID**, and so forth) as arguments to these functions, rather than supplying the hexadecimal values explicitly.

Intel x86-derived platforms support the DENORMAL input and output values in hardware. The x86 behavior is to preserve DENORMAL values. The ARM platform and the x64 platforms that have SSE2 support enable DENORMAL operands and results to be flushed, or forced to zero. The **_controlfp** and **_control87** functions provide a mask to change this behavior. The following example demonstrates the use of this mask.

```
_controlfp(_DN_SAVE, _MCW_DN);
// Denormal values preserved on ARM platforms and on x64 processors with
// SSE2 support. NOP on x86 platforms.
_controlfp(_DN_FLUSH, _MCW_DN);
// Denormal values flushed to zero by hardware on ARM platforms
// and x64 processors with SSE2 support. Ignored on other x86 platforms.
```

On ARM platforms, the **_control87** and **_controlfp** functions apply to the FPSCR register. On x64 architectures, only the SSE2 control word that's stored in the MXCSR register is affected. On x86 platforms, **_control87** and **_controlfp** affect the control words for both the x87 and the SSE2, if present. The function **__control87_2** enables both the x87 and SSE2 floating-point units to be controlled together or separately. If you want to affect both units, pass in the addresses of two integers to **x86_cw** and **sse2_cw**. If you only want to affect one unit, pass in an address for that parameter but pass in 0 (**NULL**) for the other. If 0 is passed for one of these parameters, the function has no effect on that floating-point unit. This functionality could be useful in situations where part of the code uses the x87 floating-point unit and another part of the code uses the SSE2 floating-point unit. If you use **__control87_2** in one part of a program and set different values for the floating-point control words, and then use **_control87** or **_controlfp** to further manipulate the control word, then **_control87** and **_controlfp** might be unable to return a single control word to represent the state of both floating-point units. In such a case, these functions set the **EM_AMBIGUOUS** flag in the returned integer value to indicate that there is an inconsistency between the two control words. This is a warning that the returned control word might not represent the state of both floating-point control words accurately.

On the ARM and x64 architectures, changing the infinity mode or the floating-point precision is not supported. If the precision control mask is used on the x64 platform, the function raises an assertion and the invalid parameter handler is invoked, as described in Parameter Validation.

> **NOTE**
> **__control87_2** is not supported on the ARM or x64 architectures. If you use **__control87_2** and compile your program for the ARM or x64 architectures, the compiler generates an error.

These functions are ignored when you use /clr (Common Language Runtime Compilation) to compile because the common language runtime (CLR) only supports the default floating-point precision.

Hexadecimal Values

For the **_MCW_EM** mask, clearing the mask sets the exception, which allows the hardware exception; setting the mask hides the exception. If a **_EM_UNDERFLOW** or **_EM_OVERFLOW** occurs, no hardware exception is thrown until the next floating-point instruction is executed. To generate a hardware exception immediately after **_EM_UNDERFLOW** or **_EM_OVERFLOW**, call the **FWAIT** MASM instruction.

MASK	HEX VALUE	CONSTANT	HEX VALUE
_MCW_DN (Denormal control)	0x03000000	**_DN_SAVE** **_DN_FLUSH**	0x00000000 0x01000000
_MCW_EM (Interrupt exception mask)	0x0008001F	**_EM_INVALID** **_EM_DENORMAL** **_EM_ZERODIVIDE** **_EM_OVERFLOW** **_EM_UNDERFLOW** **_EM_INEXACT**	0x00000010 0x00080000 0x00000008 0x00000004 0x00000002 0x00000001
_MCW_IC (Infinity control) (Not supported on ARM or x64] platforms.)	0x00040000	**_IC_AFFINE** **_IC_PROJECTIVE**	0x00040000 0x00000000
_MCW_RC (Rounding control)	0x00000300	**_RC_CHOP** **_RC_UP** **_RC_DOWN** **_RC_NEAR**	0x00000300 0x00000200 0x00000100 0x00000000
_MCW_PC (Precision control) (Not supported on ARM or x64 platforms.)	0x00030000	**_PC_24** (24 bits) **_PC_53** (53 bits) **_PC_64** (64 bits)	0x00020000 0x00010000 0x00000000

Requirements

ROUTINE	REQUIRED HEADER
_control87, **_controlfp**, **_control87_2**	<float.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cntrl87.c
// processor: x86
// This program uses __control87_2 to output the x87 control
// word, set the precision to 24 bits, and reset the status to
// the default.

#include <stdio.h>
#include <float.h>
#pragma fenv_access (on)

int main( void )
{
    double a = 0.1;
    unsigned int control_word_x87;

    // Show original x87 control word and do calculation.
    control_word_x87 = __control87_2(0, 0,
                                     &control_word_x87, 0);
    printf( "Original: 0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Set precision to 24 bits and recalculate.
    control_word_x87 = __control87_2(_PC_24, MCW_PC,
                                     &control_word_x87, 0);
    printf( "24-bit:   0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Restore default precision-control bits and recalculate.
    control_word_x87 = __control87_2( _CW_DEFAULT, MCW_PC,
                                     &control_word_x87, 0 );
    printf( "Default:  0x%.4x\n", control_word_x87 );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );
}
```

```
Original: 0x0001
0.1 * 0.1 = 1.000000000000000e-002
24-bit:   0x0001
0.1 * 0.1 = 9.999999776482582e-003
Default:  0x0001
0.1 * 0.1 = 1.000000000000000e-002
```

See also

Floating-Point Support
_clear87, _clearfp
_status87, _statusfp, _statusfp2
_controlfp_s

_controlfp_s

10/31/2018 • 5 minutes to read • Edit Online

Gets and sets the floating-point control word. This version of _control87, _controlfp, __control87_2 has security enhancements, as described in Security Features in the CRT.

Syntax

```
errno_t _controlfp_s(
    unsigned int *currentControl,
    unsigned int newControl,
    unsigned int mask
);
```

Parameters

currentControl
The current control-word bit value.

newControl
New control-word bit values.

mask
Mask for new control-word bits to set.

Return Value

Zero if successful, or an **errno** value error code.

Remarks

The **_controlfp_s** function is a platform-independent and more secure version of **_control87**, which gets the floating-point control word into the address that's stored in *currentControl* and sets it by using *newControl*. The bits in the values indicate the floating-point control state. The floating-point control state enables the program to change the precision, rounding, and infinity modes in the floating-point math package, depending on the platform. You can also use **_controlfp_s** to mask or unmask floating-point exceptions.

If the value for *mask* is equal to 0, **_controlfp_s** gets the floating-point control word and stores the retrieved value in *currentControl*.

If *mask* is nonzero, a new value for the control word is set: For any bit that is set (that is, equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. In other words, *fpcntrl* = ((*fpcntrl* & ~*mask*) | (*newControl* & *mask*)) where *fpcntrl* is the floating-point control word. In this scenario, *currentControl* is set to the value after the change completes; it is not the old control-word bit value.

> **NOTE**
> By default, the run-time libraries mask all floating-point exceptions.

_controlfp_s is nearly identical to the **_control87** function on Intel (x86), x64, and ARM platforms. If you are targeting x86, x64, or ARM platforms, you can use **_control87** or **_controlfp_s**.

The difference between **_control87** and **_controlfp_s** is in how they treat denormal values. For Intel (x86), x64, and ARM platforms, **_control87** can set and clear the DENORMAL OPERAND exception mask. **_controlfp_s** does not modify the DENORMAL OPERAND exception mask. This example demonstrates the difference:

```
_control87( _EM_INVALID, _MCW_EM );
// DENORMAL is unmasked by this call.
unsigned int current_word = 0;
_controlfp_s( &current_word, _EM_INVALID, _MCW_EM );
// DENORMAL exception mask remains unchanged.
```

The possible values for the mask constant (*mask*) and new control values (*newControl*) are shown in the following Hexadecimal Values table. Use the portable constants listed below (**_MCW_EM**, **_EM_INVALID**, and so on) as arguments to these functions, rather than supplying the hexadecimal values explicitly.

Intel (x86)-derived platforms support the DENORMAL input and output values in hardware. The x86 behavior is to preserve DENORMAL values. The ARM platform and the x64 platforms that have SSE2 support enable DENORMAL operands and results to be flushed, or forced to zero. The **_controlfp_s**, **_controlfp**, and **_control87** functions provide a mask to change this behavior. The following example demonstrates the use of this mask:

```
unsigned int current_word = 0;
_controlfp_s(&current_word, _DN_SAVE, _MCW_DN);
// Denormal values preserved on ARM platforms and on x64 processors with
// SSE2 support. NOP on x86 platforms.
_controlfp_s(&current_word, _DN_FLUSH, _MCW_DN);
// Denormal values flushed to zero by hardware on ARM platforms
// and x64 processors with SSE2 support. Ignored on other x86 platforms.
```

On ARM platforms, the **_controlfp_s** function applies to the FPSCR register. On x64 architectures, only the SSE2 control word that's stored in the MXCSR register is affected. On Intel (x86) platforms, **_controlfp_s** affects the control words for both the x87 and the SSE2, if present. It is possible for the two control words to be inconsistent with each other (because of a previous call to __control87_2, for example); if there is an inconsistency between the two control words, **_controlfp_s** sets the **EM_AMBIGUOUS** flag in *currentControl*. This is a warning that the returned control word might not represent the state of both floating-point control words accurately.

On the ARM and x64 architectures, changing the infinity mode or the floating-point precision is not supported. If the precision control mask is used on the x64 platform, the function raises an assertion and the invalid parameter handler is invoked, as described in Parameter Validation.

If the mask is not set correctly, this function generates an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

This function is ignored when you use /clr (Common Language Runtime Compilation) to compile because the common language runtime (CLR) only supports the default floating-point precision.

Mask constants and values

For the **_MCW_EM** mask, clearing it sets the exception, which allows the hardware exception; setting it hides the exception. If a **_EM_UNDERFLOW** or **_EM_OVERFLOW** occurs, no hardware exception is thrown until the next floating-point instruction is executed. To generate a hardware exception immediately after **_EM_UNDERFLOW** or **_EM_OVERFLOW**, call the FWAIT MASM instruction.

MASK	HEX VALUE	CONSTANT	HEX VALUE
_MCW_DN (Denormal control)	0x03000000	**_DN_SAVE** **_DN_FLUSH**	0x00000000 0x01000000

MASK	HEX VALUE	CONSTANT	HEX VALUE
_MCW_EM (Interrupt exception mask)	0x0008001F	**_EM_INVALID**	0x00000010
		_EM_DENORMAL	0x00080000
		_EM_ZERODIVIDE	0x00000008
		_EM_OVERFLOW	0x00000004
		_EM_UNDERFLOW	0x00000002
		_EM_INEXACT	0x00000001
_MCW_IC (Infinity control) (Not supported on ARM or x64 platforms.)	0x00040000	**_IC_AFFINE**	0x00040000
		_IC_PROJECTIVE	0x00000000
_MCW_RC (Rounding control)	0x00000300	**_RC_CHOP**	0x00000300
		_RC_UP	0x00000200
		_RC_DOWN	0x00000100
		_RC_NEAR	0x00000000
_MCW_PC (Precision control) (Not supported on ARM or x64 platforms.)	0x00030000	**_PC_24** (24 bits)	0x00020000
		_PC_53 (53 bits)	0x00010000
		_PC_64 (64 bits)	0x00000000

Requirements

ROUTINE	REQUIRED HEADER
_controlfp_s	<float.h>

For more compatibility information, see Compatibility.

Example

```
// crt_contrlfp_s.c
// processor: x86
// This program uses _controlfp_s to output the FP control
// word, set the precision to 24 bits, and reset the status to
// the default.

#include <stdio.h>
#include <float.h>
#pragma fenv_access (on)

int main( void )
{
    double a = 0.1;
    unsigned int control_word;
    int err;

    // Show original FP control word and do calculation.
    err = _controlfp_s(&control_word, 0, 0);
    if ( err ) /* handle error here */;

    printf( "Original: 0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Set precision to 24 bits and recalculate.
    err = _controlfp_s(&control_word, _PC_24, MCW_PC);
    if ( err ) /* handle error here */;

    printf( "24-bit:   0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );

    // Restore default precision-control bits and recalculate.
    err = _controlfp_s(&control_word, _CW_DEFAULT, MCW_PC);
    if ( err ) /* handle error here */;

    printf( "Default:  0x%.4x\n", control_word );
    printf( "%1.1f * %1.1f = %.15e\n", a, a, a * a );
}
```

```
Original: 0x9001f
0.1 * 0.1 = 1.000000000000000e-002
24-bit:   0xa001f
0.1 * 0.1 = 9.999999776482582e-003
Default:  0x9001f
0.1 * 0.1 = 1.000000000000000e-002
```

See also

Floating-Point Support
_clear87, _clearfp
_status87, _statusfp, _statusfp2
_control87, _controlfp, __control87_2

copysign, copysignf, copysignl, _copysign, _copysignf, _copysignl

10/31/2018 • 2 minutes to read • Edit Online

Returns a value that has the magnitude of one argument and the sign of another.

Syntax

```
double copysign(
   double x,
   double y
);
float copysign(
   float x,
   float y
); // C++ only
long double copysign(
   long double x,
   long double y
); // C++ only
float copysignf(
   float x,
   float y
); // C++ only
long double copysignl(
   long double x,
   long double y
); // C++ only
double _copysign(
   double x,
   double y
);
long double _copysignl(
   long double x,
   long double y
);
```

Parameters

x
The floating-point value that's returned as the magnitude of the result.

y
The floating-point value that's returned as the sign of the result.

Floating-Point Support Routines

Return Value

The **copysign** functions return a floating-point value that combines the magnitude of *x* and the sign of *y*. There is no error return.

Remarks

Because C++ allows overloading, you can call overloads of **copysign** that take and return **float** or **long double**

values. In a C program, **copysign** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER
_copysign	<float.h>
copysign, **copysignf**, **copysignl**, **_copysignf**, **_copysignl**	<math.h>

For more compatibility information, see Compatibility.

See also

fabs, fabsf, fabsl
_chgsign, _chgsignf, _chgsignl

cos, cosf, cosl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the cosine.

Syntax

```
double cos( double x );
float cosf( float x );
long double cosl( long double x );
```

```
float cos( float x );  // C++ only
long double cos( long double x );  // C++ only
```

Parameters

x
Angle in radians.

Return Value

The cosine of *x*. If *x* is greater than or equal to 263, or less than or equal to -263, a loss of significance in the result occurs.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN, IND	none	**_DOMAIN**
± INF	**INVALID**	**_DOMAIN**

Remarks

Because C++ allows overloading, you can call overloads of **cos** that take and return **float** or **long double** values. In a C program, **cos** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
cos, **cosh**, **cosf**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

See the example in sin, sinf, sinl.

See also

cosh, coshf, coshl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the hyperbolic cosine.

Syntax

```
double cosh( double x );
float coshf( float x );
long double coshl( long double x );
```

```
float cosh( float x );  // C++ only
long double cosh( long double x );  // C++ only
```

Parameters

x
Angle in radians.

Return Value

The hyperbolic cosine of *x*.

By default, if the result is too large in a **cosh**, **coshf**, or **coshl** call, the function returns **HUGE_VAL** and sets **errno** to **ERANGE**.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	none	**_DOMAIN**
x ≥ 7.104760e+002	**INEXACT**+**OVERFLOW**	**OVERFLOW**

Remarks

Because C++ allows overloading, you can call overloads of **cosh** that take and return **float** or **long double** values. In a C program, **cosh** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
coshf, **cosl**, **coshl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

See the example in sinh, sinhf, sinhl.

See also

Floating-Point Support
acosh, acoshf, acoshl
asinh, asinhf, asinhl
atanh, atanhf, atanhl
_matherr
sinh, sinhf, sinhl
tanh, tanhf, tanhl

_countof Macro

10/31/2018 • 2 minutes to read • Edit Online

Computes the number of elements in a statically-allocated array.

Syntax

```
#define _countof(array) (sizeof(array) / sizeof(array[0]))
```

Parameters

array
The name of an array.

Return Value

The number of elements in the array, expressed as a **size_t**.

Remarks

_countof is implemented as a function-like preprocessor macro. The C++ version has extra template machinery to detect at compile time if a pointer is passed instead of a statically declared array.

Ensure that *array* is actually an array, not a pointer. In C,**_countof** produces erroneous results if *array* is a pointer. In C++, **_countof** fails to compile if *array* is a pointer. An array passed as a parameter to a function *decays to a pointer*, which means that within the function, you can't use **_countof** to determine the extent of the array.

Requirements

MACRO	REQUIRED HEADER
_countof	<stdlib.h>

Example

```
// crt_countof.cpp
#define _UNICODE
#include <stdio.h>
#include <stdlib.h>
#include <tchar.h>

int main( void )
{
   _TCHAR arr[20], *p;
   printf( "sizeof(arr) = %zu bytes\n", sizeof(arr) );
   printf( "_countof(arr) = %zu elements\n", _countof(arr) );
   // In C++, the following line would generate a compile-time error:
   // printf( "%zu\n", _countof(p) ); // error C2784 (because p is a pointer)

   _tcscpy_s( arr, _countof(arr), _T("a string") );
   // unlike sizeof, _countof works here for both narrow- and wide-character strings
}
```

```
sizeof(arr) = 40 bytes
_countof(arr) = 20 elements
```

See also

sizeof Operator

cpow, cpowf, cpowl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the value of a number raised to the specified power, where the base and exponent are complex numbers. This function has a branch cut for the exponent along the negative real axis.

Syntax

```
_Dcomplex cpow(
   _Dcomplex x, _Dcomplex y
);
_Fcomplex cpow(
   _Fcomplex x, _Fcomplex y
);  // C++ only
_Lcomplex cpow(
   _Lcomplex x, _Lcomplex y
);  // C++ only
_Fcomplex cpowf(
   _Fcomplex x, _Fcomplex y
);
_Lcomplex cpowl(
   _Lcomplex x, _Lcomplex y
);
```

Parameters

x
The base.

y
The exponent.

Return Value

The value of *x* raised to the power of *y* with a branch cut for *x* along the negative real axis.

Remarks

Because C++ allows overloading, you can call overloads of **cpow** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **cpow** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cpow, **cpowf**, **cpowl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
cexp, cexpf, cexpl

clog10, clog10f, clog10l
clog, clogf, clogl

cprintf

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _cprintf or security-enhanced _cprintf_s instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_cprintf, _cprintf_l, _cwprintf, _cwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Formats and prints to the console. More-secure versions are available; see _cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cprintf(
   const char * format [, argument_list]
);
int _cprintf_l(
   const char * format,
   locale_t locale [, argument_list]
);
int _cwprintf(
   const wchar * format [, argument_list]
);
int _cwprintf_l(
   const wchar * format,
   locale_t locale [, argument_list]
);
```

Parameters

format
Format-control string.

argument_list
Optional parameters for the format string.

locale
The locale to use.

Return Value

The number of characters printed.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** function (**_putwch** for **_cwprintf**) to output characters. Each argument in *argument_list* (if any) is converted and output according to the corresponding format specification in *format*. The *format* argument uses the format specification syntax for printf and wprintf functions. Unlike the **fprintf**, **printf**, and **sprintf** functions, neither **_cprintf** nor **_cwprintf** translates line-feed characters into carriage return-line feed (CR-LF) combinations when output.

An important distinction is that **_cwprintf** displays Unicode characters when used in Windows. Unlike

_cprintf, **_cwprintf** uses the current console locale settings.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

_cprintf validates the *format* parameter. If *format* is a null pointer, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcprintf	**_cprintf**	**_cprintf**	**_cwprintf**
_tcprintf_l	**_cprintf_l**	**_cprintf_l**	**_cwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
_cprintf, **_cprintf_l**	<conio.h>
_cwprintf, **_cwprintf_l**	<conio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cprintf.c
// compile with: /c
// This program displays some variables to the console.

#include <conio.h>

int main( void )
{
    int         i = -16,
                h = 29;
    unsigned    u = 62511;
    char        c = 'A';
    char        s[] = "Test";

    // Note that console output does not translate \n as
    // standard output does. Use \r\n instead.
    //
    _cprintf( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );
}
```

```
-16  001d  62511  A Test
```

See also

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l
Format Specification Syntax: printf and wprintf Functions

_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Formats and prints to the console, and supports positional parameters in the format string.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cprintf_p(
   const char * format [,
   argument] ...
);
int _cprintf_p_l(
   const char * format,
   locale_t locale [,
   argument] ...
);
int _cwprintf_p(
   const wchar * format [,
   argument] ...
);
int _cwprintf_p_l(
   const wchar * format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional parameters.

locale
The locale to use.

Return Value

The number of characters printed or a negative value if an error occurs.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** and **_putwch** functions to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf_p function. The difference between **_cprintf_p** and **cprintf_s** is that **_cprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string.

For more information, see printf_p Positional Parameters.

Unlike the **fprintf_p**, **printf_p**, and **sprintf_p** functions, neither **_cprintf_p** nor **_cwprintf_p** translates line-feed characters into carriage return-line feed (CR-LF) combinations when output. An important distinction is that **_cwprintf_p** displays Unicode characters when used in Windows NT. Unlike **_cprintf_p**, **_cwprintf_p** uses the current console locale settings.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Also, like **_cprintf_s** and **_cwprintf_s**, they validate the input pointer and the format string. If *format* or *argument* are **NULL**, or of the format string contains invalid formatting characters, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcprintf_p	**_cprintf_p**	**_cprintf_p**	**_cwprintf_p**
_tcprintf_p_l	**_cprintf_p_l**	**_cprintf_p_l**	**_cwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_cprintf_p, **_cprintf_p_l**	<conio.h>
_cwprintf_p, **_cwprintf_p_l**	<conio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cprintf_p.c
// This program displays some variables to the console
// using the _cprintf_p function.

#include <conio.h>

int main( void )
{
    int         i = -16,
                h = 29;
    unsigned    u = 62511;
    char        c = 'A';
    char        s[] = "Test";

    // Note that console output does not translate
    // \n as standard output does. Use \r\n instead.
    _cprintf_p( "%2$d  %1$.4x  %3$u  %4$c %5$s\r\n",
                h, i, u, c, s );
}
```

```
-16  001d  62511  A Test
```

See also

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
printf_p Positional Parameters
Format Specification Syntax: printf and wprintf Functions

_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Formats and prints to the console. These versions of _cprintf, _cprintf_l, _cwprintf, _cwprintf_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cprintf_s(
   const char * format [,
   argument] ...
);
int _cprintf_s_l(
   const char * format,
   locale_t locale [,
   argument] ...
);
int _cwprintf_s(
   const wchar * format [,
   argument] ...
);
int _cwprintf_s_l(
   const wchar * format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional parameters.

locale
The locale to use.

Return Value

The number of characters printed.

Remarks

These functions format and print a series of characters and values directly to the console, using the **_putch** function (**_putwch** for **_cwprintf_s**) to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf_s function. Unlike the **fprintf_s**, **printf_s**, and **sprintf_s** functions, neither

_cprintf_s nor **_cwprintf_s** translates line-feed characters into carriage return-line feed (CR-LF) combinations when output.

An important distinction is that **_cwprintf_s** displays Unicode characters when used in Windows NT. Unlike **_cprintf_s**, **_cwprintf_s** uses the current console locale

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Like the non-secure versions (see _cprintf, _cprintf_l, _cwprintf, _cwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if *format* is a null pointer. These functions differ from the non-secure versions in that the format string itself is also validated. If there are any unknown or badly formed formatting specifiers, these functions invoke the invalid parameter handler. In all cases, If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcprintf_s	**_cprintf_s**	**_cprintf_s**	**_cwprintf_s**
_tcprintf_s_l	**_cprintf_s_l**	**_cprintf_s_l**	**_cwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_cprintf_s, **_cprintf_s_l**	<conio.h>
_cwprintf_s, **_cwprintf_s_l**	<conio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cprintf_s.c
// compile with: /c
// This program displays some variables to the console.

#include <conio.h>

int main( void )
{
   int      i = -16, h = 29;
   unsigned u = 62511;
   char     c = 'A';
   char     s[] = "Test";

   /* Note that console output does not translate \n as
    * standard output does. Use \r\n instead.
    */
   _cprintf_s( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );
}
```

```
-16  001d  62511  A Test
```

See also

Console and Port I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
printf_s, _printf_s_l, wprintf_s, _wprintf_s_l
sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l
Format Specification Syntax: printf and wprintf Functions

cproj, cprojf, cprojl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the projection of a complex number on the Reimann sphere.

Syntax

```
_Dcomplex cproj(
   _Dcomplex z
);
_Fcomplex cproj(
   _Fcomplex z
);  // C++ only
_Lcomplex cproj(
   _Lcomplex z
);  // C++ only
_Fcomplex cprojf(
   _Fcomplex z
);
_Lcomplex cprojl(
   _Lcomplex z
);
```

Parameters

z
A complex number.

Return Value

The projection of *z* on the Reimann sphere.

Remarks

Because C++ allows overloading, you can call overloads of **cproj** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **cproj** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
cproj, **cprojf**, **cprojl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
norm, normf, norml
creal, crealf, creall
conj, conjf, conjl
cimag, cimagf, cimagl

carg, cargf, cargl
cabs, cabsf, cabsl

cputs

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _cputs instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_cputs, _cputws

10/31/2018 • 2 minutes to read • Edit Online

Puts a string to the console.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cputs(
   const char *str
);
int _cputws(
   const wchar_t *str
);
```

Parameters

str
Output string.

Return Value

If successful, **_cputs** returns 0. If the function fails, it returns a nonzero value.

Remarks

The **_cputs** function writes the null-terminated string that's pointed to by *str* directly to the console. A carriage return-line feed (CR-LF) combination is not automatically appended to the string.

This function validates its parameter. If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_cputts	**_cputs**	**_cputs**	**_cputws**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_cputs	<conio.h>	<errno.h>
_cputws	<conio.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cputs.c
// compile with: /c
// This program first displays a string to the console.

#include <conio.h>
#include <errno.h>

void print_to_console(char* buffer)
{
   int retval;
   retval = _cputs( buffer );
   if (retval)
   {
       if (errno == EINVAL)
       {
         _cputs( "Invalid buffer in print_to_console.\r\n");
       }
       else
         _cputs( "Unexpected error in print_to_console.\r\n");
   }
}

void wprint_to_console(wchar_t* wbuffer)
{
   int retval;
   retval = _cputws( wbuffer );
   if (retval)
   {
       if (errno == EINVAL)
       {
         _cputws( L"Invalid buffer in wprint_to_console.\r\n");
       }
       else
         _cputws( L"Unexpected error in wprint_to_console.\r\n");
   }
}

int main()
{

   // String to print at console.
   // Notice the \r (return) character.
   char* buffer = "Hello world (courtesy of _cputs)!\r\n";
   wchar_t *wbuffer = L"Hello world (courtesy of _cputws)!\r\n";
   print_to_console(buffer);
   wprint_to_console( wbuffer );
}
```

```
Hello world (courtesy of _cputs)!
Hello world (courtesy of _cputws)!
```

See also

Console and Port I/O

_putch, _putwch

creal, crealf, creall

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the real part of a complex number.

Syntax

```
double creal( _Dcomplex z );
float crealf( _Fcomplex z );
long double creall( _Lcomplex z );
```

```
float creal( _Fcomplex z );  // C++ only
long double creal( _Lcomplex z );  // C++ only
```

Parameters

z
A complex number.

Return Value

The real part of *z*.

Remarks

Because C++ allows overloading, you can call overloads of **creal** that take **_Fcomplex** or **_Lcomplex** values, and return **float** or **long double** values. In a C program, **creal** always takes a **_Dcomplex** value and returns a **double** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
creal, **crealf**, **creall**	<complex.h>	<ccomplex>

The **_Fcomplex**, **_Dcomplex**, and **_Lcomplex** types are Microsoft-specific equivalents of the unimplemented native C99 types **float _Complex**, **double _Complex**, and **long double _Complex**, respectively. For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
_Cbuild, _FCbuild, _LCbuild
norm, normf, norml
cproj, cprojf, cprojl
conj, conjf, conjl
cimag, cimagf, cimagl
carg, cargf, cargl
cabs, cabsf, cabsl

creat

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _creat instead.

_creat, _wcreat

10/31/2018 • 2 minutes to read • Edit Online

Creates a new file. **_creat** and **_wcreat** have been deprecated; use _sopen_s, _wsopen_s instead.

Syntax

```
int _creat(
   const char *filename,
   int pmode
);
int _wcreat(
   const wchar_t *filename,
   int pmode
);
```

Parameters

filename
Name of new file.

pmode
Permission setting.

Return Value

These functions, if successful, return a file descriptor to the created file. Otherwise, the functions return -1 and set **errno** as shown in the following table.

ERRNO SETTING	DESCRIPTION
EACCES	*filename* specifies an existing read-only file or specifies a directory instead of a file.
EMFILE	No more file descriptors are available.
ENOENT	Specified file could not be found.

If *filename* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_creat** function creates a new file or opens and truncates an existing one. **_wcreat** is a wide-character version of **_creat**; the *filename* argument to **_wcreat** is a wide-character string. **_wcreat** and **_creat** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcreat	**_creat**	**_creat**	**_wcreat**

If the file specified by *filename* does not exist, a new file is created with the given permission setting and is opened for writing. If the file already exists and its permission setting allows writing, **_creat** truncates the file to length 0, destroying the previous contents, and opens it for writing. The permission setting, *pmode*, applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. The integer expression *pmode* contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, defined in SYS\Stat.h. When both constants are given, they are joined with the bitwise or operator (**|**). The *pmode* parameter is set to one of the following values.

VALUE	DEFINITION
_S_IWRITE	Writing permitted.
_S_IREAD	Reading permitted.
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

If write permission is not given, the file is read-only. All files are always readable; it is impossible to give write-only permission. The modes **_S_IWRITE** and **_S_IREAD** | **_S_IWRITE** are then equivalent. Files opened using **_creat** are always opened in compatibility mode (see _sopen) with **_SH_DENYNO**.

_creat applies the current file-permission mask to *pmode* before setting the permissions (see _umask). **_creat** is provided primarily for compatibility with previous libraries. A call to **_open** with **_O_CREAT** and **_O_TRUNC** in the *oflag* parameter is equivalent to **_creat** and is preferable for new code.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_creat	<io.h>	<sys/types.h>, <sys/stat.h>, <errno.h>
_wcreat	<io.h> or <wchar.h>	<sys/types.h>, <sys/stat.h>, <errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_creat.c
// compile with: /W3
// This program uses _creat to create
// the file (or truncate the existing file)
// named data and open it for writing.

#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int fh;

   fh = _creat( "data", _S_IREAD | _S_IWRITE ); // C4996
   // Note: _creat is deprecated; use _sopen_s instead
   if( fh == -1 )
      perror( "Couldn't create data file" );
   else
   {
      printf( "Created data file.\n" );
      _close( fh );
   }
}
```

```
Created data file.
```

See also

Low-Level I/O
_chmod, _wchmod
_chsize
_close
_dup, _dup2
_open, _wopen
_sopen, _wsopen
_umask

_create_locale, _wcreate_locale

11/8/2018 • 3 minutes to read • Edit Online

Creates a locale object.

Syntax

```
_locale_t _create_locale(
   int category,
   const char *locale
);
_locale_t _wcreate_locale(
   int category,
   const wchar_t *locale
);
```

Parameters

category
Category.

locale
Locale specifier.

Return Value

If a valid *locale* and *category* are given, returns the specified locale settings as a **_locale_t** object. The current locale settings of the program are not changed.

Remarks

The **_create_locale** function allows you to create an object that represents certain region-specific settings, for use in locale-specific versions of many CRT functions (functions with the **_l** suffix). The behavior is similar to **setlocale**, except that instead of applying the specified locale settings to the current environment, the settings are saved in a **_locale_t** structure that is returned. The **_locale_t** structure should be freed using _free_locale when it is no longer needed.

_wcreate_locale is a wide-character version of **_create_locale**; the *locale* argument to **_wcreate_locale** is a wide-character string. **_wcreate_locale** and **_create_locale** behave identically otherwise.

The *category* argument specifies the parts of the locale-specific behavior that are affected. The flags used for *category* and the parts of the program they affect are as shown in this table:

CATEGORY FLAG	AFFECTS
LC_ALL	All categories, as listed below.
LC_COLLATE	The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, **strxfrm**, **_strncoll**, **_strnicoll**, **_wcsncoll**, **_wcsnicoll**, and **wcsxfrm** functions.
LC_CTYPE	The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).

CATEGORY FLAG	AFFECTS
LC_MONETARY	Monetary-formatting information returned by the **localeconv** function.
LC_NUMERIC	Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the non-monetary formatting information returned by **localeconv**. In addition to the decimal-point character, **LC_NUMERIC** sets the thousands separator and the grouping control string returned by localeconv.
LC_TIME	The **strftime** and **wcsftime** functions.

This function validates the *category* and *locale* parameters. If the category parameter is not one of the values given in the previous table or if *locale* is **NULL**, the function returns **NULL**.

The *locale* argument is a pointer to a string that specifies the locale. For information about the format of the *locale* argument, see Locale Names, Languages, and Country/Region Strings.

The *locale* argument can take a locale name, a language string, a language string and country/region code, a code page, or a language string, country/region code, and code page. The set of available locale names, languages, country/region codes, and code pages includes all that are supported by the Windows NLS API except the code pages that require more than two bytes per character—for example, UTF-7 and UTF-8. If you provide a code page like UTF-7 or UTF-8, **_create_locale** will fail and return **NULL**. The set of locale names supported by **_create_locale** are described in Locale Names, Languages, and Country/Region Strings. The set of language and country/region strings supported by **_create_locale** are listed in Language Strings and Country/Region Strings.

For more information about locale settings, see setlocale, _wsetlocale.

The previous name of this function, **__create_locale** (with two leading underscores), has been deprecated.

Requirements

ROUTINE	REQUIRED HEADER
_create_locale	<locale.h>
_wcreate_locale	<locale.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_create_locale.c
// Sets the current locale to "de-CH" using the
// setlocale function and demonstrates its effect on the strftime
// function.

#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
    time_t ltime;
    struct tm thetime;
    unsigned char str[100];
    _locale_t locale;

    // Create a locale object representing the German (Switzerland) locale
    locale = _create_locale(LC_ALL, "de-CH");
    time (&ltime);
    _gmtime64_s(&thetime, &ltime);

    // %#x is the long date representation, appropriate to
    // the current locale
    if (!_strftime_l((char *)str, 100, "%#x",
                    (const struct tm *)&thetime, locale))
    {
        printf("_strftime_l failed!\n");
    }
    else
    {
        printf("In de-CH locale, _strftime_l returns '%s'\n", str);
    }

    _free_locale(locale);

    // Create a locale object representing the default C locale
    locale = _create_locale(LC_ALL, "C");
    time(&ltime);
    _gmtime64_s(&thetime, &ltime);

    if (!_strftime_l((char *)str, 100, "%#x",
                    (const struct tm *)&thetime, locale))
    {
        printf("_strftime_l failed!\n");
    }
    else
    {
        printf("In 'C' locale, _strftime_l returns '%s'\n", str);
    }

    _free_locale(locale);
}
```

```
In de-CH locale, _strftime_l returns 'Samstag, 9. Februar 2002'
In 'C' locale, _strftime_l returns 'Saturday, February 09, 2002'
```

See also

Locale Names, Languages, and Country/Region Strings
Language Strings
Country/Region Strings
_free_locale

_configthreadlocale
setlocale
Locale
localeconv
_mbclen, mblen, _mblen_l
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
_setmbcp
setlocale, _wsetlocale
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

_CrtCheckMemory

10/31/2018 • 2 minutes to read • Edit Online

Confirms the integrity of the memory blocks allocated in the debug heap (debug version only).

Syntax

```
int _CrtCheckMemory( void );
```

Return Value

If successful, **_CrtCheckMemory** returns TRUE; otherwise, the function returns FALSE.

Remarks

The **_CrtCheckMemory** function validates memory allocated by the debug heap manager by verifying the underlying base heap and inspecting every memory block. If an error or memory inconsistency is encountered in the underlying base heap, the debug header information, or the overwrite buffers, **_CrtCheckMemory** generates a debug report with information describing the error condition. When _DEBUG is not defined, calls to **_CrtCheckMemory** are removed during preprocessing.

The behavior of **_CrtCheckMemory** can be controlled by setting the bit fields of the _crtDbgFlag flag using the _CrtSetDbgFlag function. Turning the **_CRTDBG_CHECK_ALWAYS_DF** bit field ON results in **_CrtCheckMemory** being called every time a memory allocation operation is requested. Although this method slows down execution, it is useful for catching errors quickly. Turning the **_CRTDBG_ALLOC_MEM_DF** bit field OFF causes **_CrtCheckMemory** to not verify the heap and immediately return **TRUE**.

Because this function returns **TRUE** or **FALSE**, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if corruption is detected in the heap:

```
_ASSERTE( _CrtCheckMemory( ) );
```

For more information about how **_CrtCheckMemory** can be used with other debug functions, see Heap State Reporting Functions. For an overview of memory management and the debug heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtCheckMemory	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtCheckMemory**, see crt_dbg1.

See also

Debug Routines
_crtDbgFlag
_CrtSetDbgFlag

_CrtDbgBreak

10/31/2018 • 2 minutes to read • Edit Online

Sets a break point on a particular line of code. (Used in debug mode only.)

Syntax

```
void _CrtDbgBreak( void );
```

Return Value

There is no return value.

Remarks

The **_CrtDbgBreak** function sets a debug breakpoint on the particular line of code where the function resides. This function is used in debug mode only and is dependent on **_DEBUG** being previously defined.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Writing Your Own Debug Hook Functions.

Requirements

ROUTINE	REQUIRED HEADER
_CrtDbgBreak	<CRTDBG.h>

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines
__debugbreak

_CrtDbgReport, _CrtDbgReportW

10/31/2018 • 3 minutes to read • Edit Online

Generates a report with a debugging message and sends the report to three possible destinations (debug version only).

Syntax

```
int _CrtDbgReport(
   int reportType,
   const char *filename,
   int linenumber,
   const char *moduleName,
   const char *format [,
   argument] ...
);
int _CrtDbgReportW(
   int reportType,
   const wchar_t *filename,
   int linenumber,
   const wchar_t *moduleName,
   const wchar_t *format [,
   argument] ...
);
```

Parameters

reportType
Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

filename
Pointer to name of source file where assert/report occurred or **NULL**.

linenumber
Line number in source file where assert/report occurred or **NULL**.

moduleName
Pointer to name of module (.exe or .dll) where assert or report occurred.

format
Pointer to format-control string used to create the user message.

argument
Optional substitution arguments used by *format*.

Return Value

For all report destinations, **_CrtDbgReport** and **_CrtDbgReportW** return -1 if an error occurs and 0 if no errors are encountered. However, when the report destination is a debug message window and the user clicks the **Retry** button, these functions return 1. If the user clicks the **Abort** button in the Debug Message window, these functions immediately abort and do not return a value.

The _RPT, _RPTF debug macros call **_CrtDbgReport** to generate their debug reports. The wide-character versions of these macros as well as _ASSERT, _ASSERTE, _RPTW and _RPTFW, use **_CrtDbgReportW** to generate their debug reports. When **_CrtDbgReport** or **_CrtDbgReportW** return 1, these macros start the

debugger, provided that just-in-time (JIT) debugging is enabled.

Remarks

_CrtDbgReport and **_CrtDbgReportW** can send the debug report to three different destinations: a debug report file, a debug monitor (the Visual Studio debugger), or a debug message window. Two configuration functions, _CrtSetReportMode and _CrtSetReportFile, are used to specify the destination or destinations for each report type. These functions allow the reporting destination or destinations for each report type to be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_WARN** only be sent to the debug monitor, while a *reportType* of **_CRT_ASSERT** be sent to a debug message window and a user-defined report file.

_CrtDbgReportW is the wide-character version of **_CrtDbgReport**. All its output and string parameters are in wide-character strings; otherwise it is identical to the single-byte character version.

_CrtDbgReport and **_CrtDbgReportW** create the user message for the debug report by substituting the *argument*[**n**] arguments into the *format* string, using the same rules defined by the **printf** or **wprintf** functions. These functions then generate the debug report and determine the destination or destinations, based on the current report modes and file defined for *reportType*. When the report is sent to a debug message window, the *filename*, **lineNumber**, and *moduleName* are included in the information displayed in the window.

The following table lists the available choices for the report mode or modes and file and the resulting behavior of **_CrtDbgReport** and **_CrtDbgReportW**. These options are defined as bit flags in <crtdbg.h>.

REPORT MODE	REPORT FILE	_CRTDBGREPORT, _CRTDBGREPORTW BEHAVIOR
_CRTDBG_MODE_DEBUG	Not applicable	Writes message by using Windows OutputDebugString API.
_CRTDBG_MODE_WNDW	Not applicable	Calls Windows MessageBox API to create message box to display the message along with **Abort**, **Retry**, and **Ignore** buttons. If a user clicks **Abort**, **_CrtDbgReport** or **_CrtDbgReport** immediately aborts. If a user clicks **Retry**, it returns 1. If a user clicks **Ignore**, execution continues and **_CrtDbgReport** and **_CrtDbgReportW** return 0. Note that clicking **Ignore** when an error condition exists often results in "undefined behavior."
_CRTDBG_MODE_FILE	**__HFILE**	Writes message to user-supplied **HANDLE**, using the Windows WriteFile API and does not verify validity of file handle; the application is responsible for opening the report file and passing a valid file handle.
_CRTDBG_MODE_FILE	**_CRTDBG_FILE_STDERR**	Writes message to **stderr**.
_CRTDBG_MODE_FILE	**_CRTDBG_FILE_STDOUT**	Writes message to **stdout**.

The report can be sent to one, two, or three destinations or to no destination at all. For more information about specifying the report mode or modes and report file, see the _CrtSetReportMode and _CrtSetReportFile functions. For more information about using the debug macros and reporting functions, see Macros for

Reporting.

If your application needs more flexibility than that provided by **_CrtDbgReport** and **_CrtDbgReportW**, you can write your own reporting function and hook it into the C run-time library reporting mechanism by using the _CrtSetReportHook function.

Requirements

ROUTINE	REQUIRED HEADER
_CrtDbgReport	<crtdbg.h>
_CrtDbgReportW	<crtdbg.h>

_CrtDbgReport and **_CrtDbgReportW** are Microsoft extensions. For more information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtdbgreport.c
#include <crtdbg.h>

int main(int argc, char *argv[]) {
#ifdef _DEBUG
   _CrtDbgReport(_CRT_ASSERT, __FILE__, __LINE__, argv[0], NULL);
#endif
}
```

See crt_dbg2 for an example of how to change the report function.

See also

Debug Routines
_CrtSetReportMode
_CrtSetReportFile
printf, _printf_l, wprintf, _wprintf_l
_DEBUG

_CrtDoForAllClientObjects

10/31/2018 • 2 minutes to read • Edit Online

Calls an application-supplied function for all **_CLIENT_BLOCK** types in the heap (debug version only).

Syntax

```
void _CrtDoForAllClientObjects(
   void ( * pfn )( void *, void * ),
   void *context
);
```

Parameters

pfn
Pointer to the application-supplied function callback function. The first parameter to this function points to the data. The second parameter is the context pointer that is passed to the call to **_CrtDoForAllClientObjects**.

context
Pointer to the application-supplied context to pass to the application-supplied function.

Remarks

The **_CrtDoForAllClientObjects** function searches the heap's linked list for memory blocks with the **_CLIENT_BLOCK** type and calls the application-supplied function when a block of this type is found. The found block and the *context* parameter are passed as arguments to the application-supplied function. During debugging, an application can track a specific group of allocations by explicitly calling the debug heap functions to allocate the memory and specifying that the blocks be assigned the **_CLIENT_BLOCK** block type. These blocks can then be tracked separately and reported on differently during leak detection and memory state reporting.

If the **_CRTDBG_ALLOC_MEM_DF** bit field of the _crtDbgFlag flag is not turned on, **_CrtDoForAllClientObjects** immediately returns. When _DEBUG is not defined, calls to **_CrtDoForAllClientObjects** are removed during preprocessing.

For more information about the **_CLIENT_BLOCK** type and how it can be used by other debug functions, see Types of blocks on the debug heap. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

If *pfn* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns.

Requirements

ROUTINE	REQUIRED HEADER
_CrtDoForAllClientObjects	<crtdbg.h>, <errno.h>

For more compatibility information, see Compatibility.

Libraries: Debug versions of universal C run-time libraries only.

See also

Debug Routines
_CrtSetDbgFlag
Heap State Reporting Functions
_CrtReportBlockType

_CrtDumpMemoryLeaks

10/31/2018 • 2 minutes to read • Edit Online

Dumps all the memory blocks in the debug heap when a memory leak has occurred (debug version only).

Syntax

```
int _CrtDumpMemoryLeaks( void );
```

Return Value

_CrtDumpMemoryLeaks returns TRUE if a memory leak is found. Otherwise, the function returns FALSE.

Remarks

The **_CrtDumpMemoryLeaks** function determines whether a memory leak has occurred since the start of program execution. When a leak is found, the debug header information for all the objects in the heap is dumped in a user-readable form. When _DEBUG is not defined, calls to **_CrtDumpMemoryLeaks** are removed during preprocessing.

_CrtDumpMemoryLeaks is frequently called at the end of program execution to verify that all memory allocated by the application has been freed. The function can be called automatically at program termination by turning on the **_CRTDBG_LEAK_CHECK_DF** bit field of the _crtDbgFlag flag using the _CrtSetDbgFlag function.

_CrtDumpMemoryLeaks calls _CrtMemCheckpoint to obtain the current state of the heap and then scans the state for blocks that have not been freed. When an unfreed block is encountered, **_CrtDumpMemoryLeaks** calls _CrtMemDumpAllObjectsSince to dump information for all the objects allocated in the heap from the start of program execution.

By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory dump operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks in the leak detection process.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtDumpMemoryLeaks	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtDumpMemoryLeaks**, see crt_dbg1.

See also

Debug Routines

_CrtGetAllocHook

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the current client-defined allocation function for hooking into the C run-time debug memory allocation process (debug version only).

Syntax

```
_CRT_ALLOC_HOOK _CrtGetAllocHook( void );
```

Return Value

Returns the currently defined allocation hook function.

Remarks

_CrtGetAllocHook retrieves the current client-defined application hook function for the C run-time debug library memory allocation process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Debug Hook Function Writing.

Requirements

ROUTINE	REQUIRED HEADER
_CrtGetAllocHook	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines
_CrtSetAllocHook

_CrtGetDumpClient

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the current application-defined function for dumping the **_CLIENT_BLOCK** type memory blocks (debug version only).

Syntax

```
_CRT_DUMP_CLIENT _CrtGetDumpClient( void );
```

Return Value

Returns the current dump routine.

Remarks

The **_CrtGetDumpClient** function retrieves the current hook function for dumping objects stored in the **_CLIENT_BLOCK** memory blocks for the C run-time debug memory dump process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Debug Hook Function Writing.

Requirements

ROUTINE	REQUIRED HEADER
_CrtGetDumpClient	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines
_CrtReportBlockType
_CrtSetDumpClient

_CrtGetReportHook

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the client-defined reporting function for hooking it into the C run time for the debug reporting process (debug version only).

Syntax

```
_CRT_REPORT_HOOK _CrtGetReportHook( void );
```

Return Value

Returns the current client-defined reporting function.

Remarks

_CrtGetReportHook allows an application to retrieve the current reporting function for the C run-time debug library reporting process.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Debug Hook Function Writing.

Requirements

ROUTINE	REQUIRED HEADER
_CrtGetReportHook	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetReportHook**, see report.

See also

Debug Routines
_CrtSetReportHook

_CrtIsMemoryBlock

10/31/2018 • 2 minutes to read • Edit Online

Verifies that a specified memory block is in the local heap and that it has a valid debug heap block type identifier (debug version only).

Syntax

```
int _CrtIsMemoryBlock(
   const void *userData,
   unsigned int size,
   long *requestNumber,
   char **filename,
   int *linenumber
);
```

Parameters

userData
Pointer to the beginning of the memory block to verify.

size
Size of the specified block (in bytes).

requestNumber
Pointer to the allocation number of the block or **NULL**.

filename
Pointer to the name of the source file that requested the block or **NULL**.

linenumber
Pointer to the line number in the source file or **NULL**.

Return Value

_CrtIsMemoryBlock returns **TRUE** if the specified memory block is located within the local heap and has a valid debug heap block type identifier; otherwise, the function returns **FALSE**.

Remarks

The **_CrtIsMemoryBlock** function verifies that a specified memory block is located within the application's local heap and that it has a valid block type identifier. This function can also be used to obtain the object allocation order number and the source file name/line number where the memory block allocation was originally requested. Passing non-**NULL** values for the *requestNumber*, *filename*, or *linenumber* parameters causes **_CrtIsMemoryBlock** to set these parameters to the values in the memory block's debug header, if it finds the block in the local heap. When _DEBUG is not defined, calls to **_CrtIsMemoryBlock** are removed during preprocessing.

If **_CrtIsMemoryBlock** fails, it returns **FALSE** and the output parameters are initialized to default values: *requestNumber* and **lineNumber** are set to 0 and *filename* is set to **NULL**.

Because this function returns **TRUE** or **FALSE**, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the specified address is

not located within the local heap:

```
_ASSERTE( _CrtIsMemoryBlock( userData, size, &requestNumber,
          &filename, &linenumber ) );
```

For more information about how **_CrtIsMemoryBlock** can be used with other debug functions and macros, see Macros for Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtIsMemoryBlock	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

See the example for the _CrtIsValidHeapPointer topic.

See also

Debug Routines

_CrtIsValidHeapPointer

10/31/2018 • 3 minutes to read • Edit Online

Verifies that a specified pointer is in a heap allocated by some C run-time library, but not necessarily by the caller's CRT library. In versions of the CRT before Visual Studio 2010, this verifies that the specified pointer is in the local heap (debug version only).

Syntax

```
int _CrtIsValidHeapPointer(
   const void *userData
);
```

Parameters

userData
Pointer to the beginning of an allocated memory block.

Return Value

_CrtIsValidHeapPointer returns TRUE if the specified pointer is in the heap shared by all CRT library instances. In versions of the CRT before Visual Studio 2010, this returns TRUE if the specified pointer is in the local heap. Otherwise, the function returns FALSE.

Remarks

We do not recommend that you use this function. Starting with the Visual Studio 2010 CRT library, all CRT libraries share one OS heap, the *process heap*. The **_CrtIsValidHeapPointer** function reports whether the pointer was allocated in a CRT heap, but not that it was allocated by the caller's CRT library. For example, consider a block allocated by using the Visual Studio 2010 version of the CRT library. If the **_CrtIsValidHeapPointer** function exported by the Visual Studio 2012 version of the CRT library tests the pointer, it returns TRUE. This is no longer a useful test. In versions of the CRT library before Visual Studio 2010, the function is used to ensure that a specific memory address is within the local heap. The local heap refers to the heap created and managed by a particular instance of the C run-time library. If a dynamic-link library (DLL) contains a static link to the run-time library, it has its own instance of the run-time heap, and therefore its own heap, independent of the application's local heap. When _DEBUG is not defined, calls to **_CrtIsValidHeapPointer** are removed during preprocessing.

Because this function returns TRUE or FALSE, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the specified address is not located within the local heap:

```
_ASSERTE( _CrtIsValidHeapPointer( userData ) );
```

For more information about how **_CrtIsValidHeapPointer** can be used with other debug functions and macros, see Macros for Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtIsValidHeapPointer	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

The following example demonstrates how to test whether memory is valid when it is used with C run-time libraries before Visual Studio 2010. This example is provided for users of legacy CRT library code.

```
// crt_isvalid.c
// This program allocates a block of memory using _malloc_dbg
// and then tests the validity of this memory by calling
// _CrtIsMemoryBlock,_CrtIsValidPointer, and _CrtIsValidHeapPointer.

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <crtdbg.h>

#define  TRUE   1
#define  FALSE  0

int main( void )
{
    char *my_pointer;

    // Call _malloc_dbg to include the filename and line number
    // of our allocation request in the header information
    my_pointer = (char *)_malloc_dbg( sizeof(char) * 10,
        _NORMAL_BLOCK, __FILE__, __LINE__ );

    // Ensure that the memory got allocated correctly
    _CrtIsMemoryBlock((const void *)my_pointer, sizeof(char) * 10,
        NULL, NULL, NULL );

    // Test for read/write accessibility
    if (_CrtIsValidPointer((const void *)my_pointer,
        sizeof(char) * 10, TRUE))
        printf("my_pointer has read and write accessibility.\n");
    else
        printf("my_pointer only has read access.\n");

    // Make sure my_pointer is within the local heap
    if (_CrtIsValidHeapPointer((const void *)my_pointer))
        printf("my_pointer is within the local heap.\n");
    else
        printf("my_pointer is not located within the local"
               " heap.\n");

    free(my_pointer);
}
```

```
my_pointer has read and write accessibility.
my_pointer is within the local heap.
```

See also

Debug Routines

_CrtIsValidPointer

10/31/2018 • 2 minutes to read • Edit Online

Verifies that a pointer is not null. In versions of the C run-time library before Visual Studio 2010, verifies that a specified memory range is valid for reading and writing (debug version only).

Syntax

```
int _CrtIsValidPointer(
   const void *address,
   unsigned int size,
   int access
);
```

Parameters

address
Points to the beginning of the memory range to test for validity.

size
Size of the specified memory range (in bytes).

access
Read/write accessibility to determine for the memory range.

Return Value

_CrtIsValidPointer returns TRUE if the specified pointer is not null. In CRT library versions before Visual Studio 2010, returns TRUE if the memory range is valid for the specified operation or operations. Otherwise, the function returns FALSE.

Remarks

Starting with the CRT library in Visual Studio 2010, the *size* and *access* parameters are ignored, and **_CrtIsValidPointer** only verifies that the specified *address* is not null. Because this test is easy to perform yourself, we do not recommend you use this function. In versions before Visual Studio 2010, the function verifies that the memory range beginning at *address* and extending for *size* bytes is valid for the specified accessibility operation or operations. When *access* is set to TRUE, the memory range is verified for both reading and writing. When *access* is FALSE, the memory range is only validated for reading. When _DEBUG is not defined, calls to **_CrtIsValidPointer** are removed during preprocessing.

Because this function returns TRUE or FALSE, it can be passed to one of the _ASSERT macros to create a simple debugging error handling mechanism. The following example causes an assertion failure if the memory range is not valid for both reading and writing operations:

```
_ASSERTE( _CrtIsValidPointer( address, size, TRUE ) );
```

For more information about how **_CrtIsValidPointer** can be used with other debug functions and macros, see Macros for Reporting. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtIsValidPointer	<crtdbg.h>

_CrtIsValidPointer is a Microsoft extension. For compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

See the example for the _CrtIsValidHeapPointer topic.

See also

Debug Routines

_CrtMemCheckpoint

10/31/2018 • 2 minutes to read • Edit Online

Obtains the current state of the debug heap and stores in an application-supplied **_CrtMemState** structure (debug version only).

Syntax

```
void _CrtMemCheckpoint(
   _CrtMemState *state
);
```

Parameters

state
Pointer to **_CrtMemState** structure to fill with the memory checkpoint.

Remarks

The **_CrtMemCheckpoint** function creates a snapshot of the current state of the debug heap at any given moment. This snapshot can be used by other heap state functions such as _CrtMemDifference to help detect memory leaks and other problems. When _DEBUG is not defined, calls to **_CrtMemState** are removed during preprocessing.

The application must pass a pointer to a previously allocated instance of the **_CrtMemState** structure, defined in Crtdbg.h, in the *state* parameter. If **_CrtMemCheckpoint** encounters an error during the checkpoint creation, the function generates a **_CRT_WARN** debug report describing the problem.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

If *state* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns.

Requirements

ROUTINE	REQUIRED HEADER
_CrtMemCheckpoint	<crtdbg.h>, <errno.h>

For more compatibility information, see Compatibility.

Libraries: Debug versions of the UCRT only.

See also

Debug Routines
_CrtMemDifference

_CrtMemDifference

10/31/2018 • 2 minutes to read • Edit Online

Compares two memory states and returns their differences (debug version only).

Syntax

```
int _CrtMemDifference(
   _CrtMemState *stateDiff,
   const _CrtMemState *oldState,
   const _CrtMemState *newState
);
```

Parameters

stateDiff
Pointer to a **_CrtMemState** structure that is used to store the differences between the two memory states (returned).

oldState
Pointer to an earlier memory state (**_CrtMemState** structure).

newState
Pointer to a later memory state (**_CrtMemState** structure).

Return Value

If the memory states are significantly different, **_CrtMemDifference** returns TRUE. Otherwise, the function returns FALSE.

Remarks

The **_CrtMemDifference** function compares *oldState* and *newState* and stores their differences in *stateDiff*, which can then be used by the application to detect memory leaks and other memory problems. When _DEBUG is not defined, calls to **_CrtMemDifference** are removed during preprocessing.

newState and *oldState* must each be a valid pointer to a **_CrtMemState** structure, defined in Crtdbg.h, that has been filled in by _CrtMemCheckpoint before calling **_CrtMemDifference**. *stateDiff* must be a pointer to a previously allocated instance of the **_CrtMemState** structure. If *stateDiff*, *newState*, or *oldState* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, errno, _doserrno, _sys_errlist, and _sys_nerr is set to **EINVAL** and the function returns FALSE.

_CrtMemDifference compares the **_CrtMemState** field values of the blocks in *oldState* to those in *newState* and stores the result in *stateDiff*. When the number of allocated block types or total number of allocated blocks for each type differs between the two memory states, the states are said to be significantly different. The difference between the largest amount ever allocated at once for the two states and the difference between total allocations for the two states are also stored in *stateDiff*.

By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory state operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks in leak detection and other memory state operations. Freed memory blocks (**_FREE_BLOCK**) do not cause **_CrtMemDifference** to return TRUE.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_CrtMemDifference	<crtdbg.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries: Debug versions of CRT Library Features only.

See also

Debug Routines
_crtDbgFlag

_CrtMemDumpAllObjectsSince

10/31/2018 • 2 minutes to read • Edit Online

Dumps information about objects in the heap from the start of program execution or from a specified heap state (debug version only).

Syntax

```
void _CrtMemDumpAllObjectsSince(
   const _CrtMemState *state
);
```

Parameters

state
Pointer to the heap state to begin dumping from or **NULL**.

Remarks

The **_CrtMemDumpAllObjectsSince** function dumps the debug header information of objects allocated in the heap in a user-readable form. The dump information can be used by the application to track allocations and detect memory problems. When _DEBUG is not defined, calls to **_CrtMemDumpAllObjectsSince** are removed during preprocessing.

_CrtMemDumpAllObjectsSince uses the value of the *state* parameter to determine where to initiate the dump operation. To begin dumping from a specified heap state, the *state* parameter must be a pointer to a **_CrtMemState** structure that has been filled in by _CrtMemCheckpoint before **_CrtMemDumpAllObjectsSince** was called. When *state* is **NULL**, the function begins the dump from the start of program execution.

If the application has installed a dump hook function by calling _CrtSetDumpClient, then every time **_CrtMemDumpAllObjectsSince** dumps information about a **_CLIENT_BLOCK** type of block, it calls the application-supplied dump function as well. By default, internal C run-time blocks (**_CRT_BLOCK**) are not included in memory dump operations. The _CrtSetDbgFlag function can be used to turn on the **_CRTDBG_CHECK_CRT_DF** bit of **_crtDbgFlag** to include these blocks. In addition, blocks marked as freed or ignored (**_FREE_BLOCK**, **_IGNORE_BLOCK**) are not included in the memory dump.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtMemDumpAll-ObjectsSince	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtMemDumpAllObjectsSince**, see crt_dbg2.

See also

Debug Routines
_crtDbgFlag

_CrtMemDumpStatistics

10/31/2018 • 2 minutes to read • Edit Online

Dumps the debug header information for a specified heap state in a user-readable form (debug version only).

Syntax

```
void _CrtMemDumpStatistics(
   const _CrtMemState *state
);
```

Parameters

state
Pointer to the heap state to dump.

Remarks

The **_CrtMemDumpStatistics** function dumps the debug header information for a specified state of the heap in a user-readable form. The dump statistics can be used by the application to track allocations and detect memory problems. The memory state can contain a specific heap state or the difference between two states. When _DEBUG is not defined, calls to **_CrtMemDumpStatistics** are removed during preprocessing.

The *state* parameter must be a pointer to a **_CrtMemState** structure that has been filled in by _CrtMemCheckpoint or returned by _CrtMemDifference before **_CrtMemDumpStatistics** is called. If *state* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and no action is taken. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For more information about heap state functions and the **_CrtMemState** structure, see Heap State Reporting Functions. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_CrtMemDumpStatistics	<crtdbg.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries: Debug versions of CRT Library Features only.

See also

Debug Routines

_CrtReportBlockType

10/31/2018 • 2 minutes to read • Edit Online

Returns the block type/subtype associated with a given debug heap block pointer.

Syntax

```
int _CrtReportBlockType(
   const void * pBlock
);
```

Parameters

pBlock
Pointer to a valid debug heap block.

Return Value

When passed a valid debug heap pointer, the **_CrtReportBlockType** function returns the block type and subtype in the form of an **int**. When passed an invalid pointer, the function returns -1.

Remarks

To extract the type and subtype returned by **_CrtReportBlockType**, use the macros **_BLOCK_TYPE** and **_BLOCK_SUBTYPE** (both defined in Crtdbg.h) on the return value.

For information about the allocation block types and how they are used, see Types of Blocks on the Debug Heap.

Requirements

ROUTINE	REQUIRED HEADER
_CrtReportBlockType	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtreportblocktype.cpp
// compile with: /MDd

#include <malloc.h>
#include <stdio.h>
#include <crtdbg.h>

void __cdecl Dumper(void *ptr, void *)
{
    int block = _CrtReportBlockType(ptr);
    _RPT3(_CRT_WARN, "Dumper found block at %p: type %d, subtype %d\n", ptr,
          _BLOCK_TYPE(block), _BLOCK_SUBTYPE(block));
}

void __cdecl LeakDumper(void *ptr, size_t sz)
{
    int block = _CrtReportBlockType(ptr);
    _RPT4(_CRT_WARN, "LeakDumper found block at %p:"
                     " type %d, subtype %d, size %d\n", ptr,
          _BLOCK_TYPE(block), _BLOCK_SUBTYPE(block), sz);
}

int main(void)
{
    _CrtSetDbgFlag(_CrtSetDbgFlag(_CRTDBG_REPORT_FLAG) |
    _CRTDBG_LEAK_CHECK_DF);
    _CrtSetReportMode( _CRT_WARN, _CRTDBG_MODE_FILE );
    _CrtSetReportFile( _CRT_WARN, _CRTDBG_FILE_STDOUT );
    _malloc_dbg(10, _NORMAL_BLOCK , __FILE__, __LINE__);
    _malloc_dbg(10, _CLIENT_BLOCK | (1 << 16), __FILE__, __LINE__);
    _malloc_dbg(20, _CLIENT_BLOCK | (2 << 16), __FILE__, __LINE__);
    _malloc_dbg(30, _CLIENT_BLOCK | (3 << 16), __FILE__, __LINE__);
    _CrtDoForAllClientObjects(Dumper, NULL);
    _CrtSetDumpClient(LeakDumper);
}
```

Sample Output

```
Dumper found block at 00314F78: type 4, subtype 3
Dumper found block at 00314F38: type 4, subtype 2
Dumper found block at 00314F00: type 4, subtype 1
Detected memory leaks!
Dumping objects ->
crt_crtreportblocktype.cpp(30) : {55} client block at 0x00314F78, subtype 3, 30 bytes long.
Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(29) : {54} client block at 0x00314F38, subtype 2, 20 bytes long.
Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(28) : {53} client block at 0x00314F00, subtype 1, 10 bytes long.
Data: <          > CD CD CD CD CD CD CD CD CD CD
crt_crtreportblocktype.cpp(27) : {52} normal block at 0x00314EC8, 10 bytes long.
Data: <          > CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

See also

_CrtDoForAllClientObjects
_CrtSetDumpClient
_CrtMemDumpAllObjectsSince
_CrtDumpMemoryLeaks

_CrtSetAllocHook

10/31/2018 • 2 minutes to read • Edit Online

Installs a client-defined allocation function by hooking it into the C run-time debug memory allocation process (debug version only).

Syntax

```
_CRT_ALLOC_HOOK _CrtSetAllocHook(
   _CRT_ALLOC_HOOK allocHook
);
```

Parameters

allocHook
New client-defined allocation function to hook into the C run-time debug memory allocation process.

Return Value

Returns the previously defined allocation hook function, or **NULL** if *allocHook* is **NULL**.

Remarks

_CrtSetAllocHook allows an application to hook its own allocation function into the C run-time debug library memory allocation process. As a result, every call to a debug allocation function to allocate, reallocate, or free a memory block triggers a call to the application's hook function. **_CrtSetAllocHook** provides an application with an easy method for testing how the application handles insufficient memory situations, the ability to examine allocation patterns, and the opportunity to log allocation information for later analysis. When _DEBUG is not defined, calls to **_CrtSetAllocHook** are removed during preprocessing.

The **_CrtSetAllocHook** function installs the new client-defined allocation function specified in *allocHook* and returns the previously defined hook function. The following example demonstrates how a client-defined allocation hook should be prototyped:

```
int YourAllocHook( int allocType, void *userData, size_t size,
                   int blockType, long requestNumber,
                   const unsigned char *filename, int lineNumber);
```

The **allocType** argument specifies the type of allocation operation (**_HOOK_ALLOC**, **_HOOK_REALLOC**, and **_HOOK_FREE**) that triggered the call to the allocation's hook function. When the triggering allocation type is **_HOOK_FREE**, *userData* is a pointer to the user data section of the memory block about to be freed. However, when the triggering allocation type is **_HOOK_ALLOC** or **_HOOK_REALLOC**, *userData* is **NULL** because the memory block has not been allocated yet.

size specifies the size of the memory block in bytes, *blockType* indicates the type of the memory block, *requestNumber* is the object allocation order number of the memory block, and, if available, *filename* and **lineNumber** specify the source file name and line number where the triggering allocation operation was initiated.

After the hook function has finished processing, it must return a Boolean value, which tells the main C run-time allocation process how to continue. When the hook function wants the main allocation process to continue as if the hook function had never been called, then the hook function should return **TRUE**. This causes the original

triggering allocation operation to be executed. Using this implementation, the hook function can gather and save allocation information for later analysis, without interfering with the current allocation operation or state of the debug heap.

When the hook function wants the main allocation process to continue as if the triggering allocation operation was called and it failed, then the hook function should return **FALSE**. Using this implementation, the hook function can simulate a wide range of memory conditions and debug heap states to test how the application handles each situation.

To clear the hook function, pass **NULL** to **_CrtSetAllocHook**.

For more information about how **_CrtSetAllocHook** can be used with other memory management functions or how to write your own client-defined hook functions, see Debug Hook Function Writing.

> **NOTE**
> **_CrtSetAllocHook** is not supported under **/clr:pure**. The **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and removed in Visual Studio 2017.

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetAllocHook	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_CrtSetAllocHook**, see crt_dbg2.

See also

Debug Routines
_CrtGetAllocHook

_CrtSetBreakAlloc

10/31/2018 • 2 minutes to read • Edit Online

Sets a breakpoint on a specified object allocation order number (debug version only).

Syntax

```
long _CrtSetBreakAlloc(
   long lBreakAlloc
);
```

Parameters

lBreakAlloc
Allocation order number, for which to set the breakpoint.

Return Value

Returns the previous object allocation order number that had a breakpoint set.

Remarks

_CrtSetBreakAlloc allows an application to perform memory leak detection by breaking at a specific point of memory allocation and tracing back to the origin of the request. The function uses the sequential object allocation order number assigned to the memory block when it was allocated in the heap. When _DEBUG is not defined, calls to **_CrtSetBreakAlloc** are removed during preprocessing.

The object allocation order number is stored in the *lRequest* field of the **_CrtMemBlockHeader** structure, defined in Crtdbg.h. When information about a memory block is reported by one of the debug dump functions, this number is enclosed in braces, such as {36}.

For more information about how **_CrtSetBreakAlloc** can be used with other memory management functions, see Tracking Heap Allocation Requests. For more information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details.

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetBreakAlloc	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_setbrkal.c
// compile with: -D_DEBUG /MTd -Od -Zi -W3 /c /link -verbose:lib -debug

/*
* In this program, a call is made to the _CrtSetBreakAlloc routine
* to verify that the debugger halts program execution when it reaches
* a specified allocation number.
*/

#include <malloc.h>
#include <crtdbg.h>

int main( )
{
        long allocReqNum;
        char *my_pointer;

        /*
         * Allocate "my_pointer" for the first
         * time and ensure that it gets allocated correctly
         */
        my_pointer = malloc(10);
        _CrtIsMemoryBlock(my_pointer, 10, &allocReqNum, NULL, NULL);

        /*
         * Set a breakpoint on the allocation request
         * number for "my_pointer"
         */
        _CrtSetBreakAlloc(allocReqNum+2);

        /*
         * Alternate freeing and reallocating "my_pointer"
         * to verify that the debugger halts program execution
         * when it reaches the allocation request
         */
        free(my_pointer);
        my_pointer = malloc(10);
        free(my_pointer);
        my_pointer = malloc(10);
        free(my_pointer);
}
```

See also

Debug Routines

_CrtSetDbgFlag

11/12/2018 • 6 minutes to read • Edit Online

Retrieves or modifies the state of the **_crtDbgFlag** flag to control the allocation behavior of the debug heap manager (debug version only).

Syntax

```
int _CrtSetDbgFlag(
   int newFlag
);
```

Parameters

newFlag
New state for **_crtDbgFlag**.

Return Value

Returns the previous state of **_crtDbgFlag**.

Remarks

The **_CrtSetDbgFlag** function allows the application to control how the debug heap manager tracks memory allocations by modifying the bit fields of the **_crtDbgFlag** flag. By setting the bits (turning on), the application can instruct the debug heap manager to perform special debugging operations, including checking for memory leaks when the application exits and reporting if any are found, simulating low-memory conditions by specifying that freed memory blocks should remain in the heap's linked list, and verifying the integrity of the heap by inspecting each memory block at every allocation request. When _DEBUG is not defined, calls to **_CrtSetDbgFlag** are removed during preprocessing.

The following table lists the bit fields for **_crtDbgFlag** and describes their behavior. Because setting the bits results in increased diagnostic output and reduced program execution speed, these bits are not set (turned off) by default. For more information about these bit fields, see Heap State Reporting Functions.

BIT FIELD	DEFAULT	DESCRIPTION
_CRTDBG_ALLOC_MEM_DF	ON	ON: Enable debug heap allocations and use of memory block type identifiers, such as **_CLIENT_BLOCK**. OFF: Add new allocations to heap's linked list, but set block type to **_IGNORE_BLOCK**. Can also be combined with any of the heap-frequency check macros.

BIT FIELD	DEFAULT	DESCRIPTION
_CRTDBG_CHECK_ALWAYS_DF	OFF	ON: Call _CrtCheckMemory at every allocation and deallocation request. OFF: **_CrtCheckMemory** must be called explicitly. Heap-frequency check macros have no effect when this flag is set.
_CRTDBG_CHECK_CRT_DF	OFF	ON: Include **_CRT_BLOCK** types in leak detection and memory state difference operations. OFF: Memory used internally by the run-time library is ignored by these operations. Can also be combined with any of the heap-frequency check macros.
_CRTDBG_DELAY_FREE_MEM_DF	OFF	ON: Keep freed memory blocks in the heap's linked list, assign them the **_FREE_BLOCK** type, and fill them with the byte value 0xDD. OFF: Do not keep freed blocks in the heap's linked list. Can also be combined with any of the heap-frequency check macros.
_CRTDBG_LEAK_CHECK_DF	OFF	ON: Perform automatic leak checking at program exit through a call to _CrtDumpMemoryLeaks and generate an error report if the application failed to free all the memory it allocated. OFF: Do not automatically perform leak checking at program exit. Can also be combined with any of the heap-frequency check macros.

Heap-Check Frequency Macros

You can specify how often the C run-time library performs validation of the debug heap (**_CrtCheckMemory**) based on the number of calls to **malloc**, **realloc**, **free**, and **_msize**.

_CrtSetDbgFlag then inspects the upper 16 bits of the *newFlag* parameter for a value. The value specified is the number of **malloc**, **realloc**, **free**, and **_msize** calls between **_CrtCheckMemory** calls. Four predefined macros are provided for this purpose.

MACRO	NUMBER OF MALLOC, REALLOC, FREE, AND _MSIZE CALLS BETWEEN CALLS TO _CRTCHECKMEMORY
_CRTDBG_CHECK_EVERY_16_DF	16
_CRTDBG_CHECK_EVERY_128_DF	128
_CRTDBG_CHECK_EVERY_1024_DF	1024
_CRTDBG_CHECK_DEFAULT_DF	0 (by default, no heap checks)

By default, **_CrtCheckMemory** is called once every 1,024 times you call **malloc**, **realloc**, **free**, and **_msize**.

For example, you could specify a heap check every 16 **malloc**, **realloc**, **free**, and **_msize** operations with the following code:

```
#include <crtdbg.h>
int main( )
{
    int tmp;

    // Get the current bits
    tmp = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);

    // Clear the upper 16 bits and OR in the desired frequency
    tmp = (tmp & 0x0000FFFF) | _CRTDBG_CHECK_EVERY_16_DF;

    // Set the new bits
    _CrtSetDbgFlag(tmp);
}
```

The upper 16 bits of the *newFlag* parameter are ignored when _CRTDBG_CHECK_ALWAYS_DF is specified. In this case, **_CrtCheckMemory** is called each time you call **malloc**, **realloc**, **free**, and **_msize**.

newFlag is the new state to apply to the **_crtDbgFlag** and is a combination of the values for each of the bit fields.

To change one or more of these bit fields and create a new state for the flag

1. Call **_CrtSetDbgFlag** with *newFlag* equal to **_CRTDBG_REPORT_FLAG** to obtain the current **_crtDbgFlag** state and store the returned value in a temporary variable.
2. Turn on any bits by a bitwise **OR** of the temporary variable with the corresponding bitmasks (represented in the application code by manifest constants).
3. Turn off the other bits by **AND**-ing the variable with a bitwise **NOT** of the appropriate bitmasks.
4. Call **_CrtSetDbgFlag** with *newFlag* equal to the value stored in the temporary variable to set the new state for **_crtDbgFlag**.

The following code demonstrates how to simulate low-memory conditions by keeping freed memory blocks in the heap's linked list and prevent **_CrtCheckMemory** from being called at every allocation request:

```
// Get the current state of the flag
// and store it in a temporary variable
int tmpFlag = _CrtSetDbgFlag( _CRTDBG_REPORT_FLAG );

// Turn On (OR) - Keep freed memory blocks in the
// heap's linked list and mark them as freed
tmpFlag |= _CRTDBG_DELAY_FREE_MEM_DF;

// Turn Off (AND) - prevent _CrtCheckMemory from
// being called at every allocation request
tmpFlag &= ~_CRTDBG_CHECK_ALWAYS_DF;

// Set the new state for the flag
_CrtSetDbgFlag( tmpFlag );
```

For an overview of memory management and the debug heap, see CRT Debug Heap Details.

To disable a flag with the **_CrtSetDbgFlag** function, you should **AND** the variable with the bitwise **NOT** of the bitmask.

If *newFlag* is not a valid value, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns the previous state of **_crtDbgFlag**.

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetDbgFlag	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_crtsetdflag.c
// compile with: /c -D_DEBUG /MTd -Od -Zi -W3 /link -verbose:lib /debug

// This program concentrates on allocating and freeing memory
// blocks to test the functionality of the _crtDbgFlag flag.

#include <string.h>
#include <malloc.h>
#include <crtdbg.h>

int main( )
{
    char *p1, *p2;
    int tmpDbgFlag;

    _CrtSetReportMode( _CRT_ERROR, _CRTDBG_MODE_FILE );
    _CrtSetReportFile( _CRT_ERROR, _CRTDBG_FILE_STDERR );

    // Set the debug-heap flag to keep freed blocks in the
    // heap's linked list - This will allow us to catch any
    // inadvertent use of freed memory
    tmpDbgFlag = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);
    tmpDbgFlag |= _CRTDBG_DELAY_FREE_MEM_DF;
    tmpDbgFlag |= _CRTDBG_LEAK_CHECK_DF;
    _CrtSetDbgFlag(tmpDbgFlag);

    // Allocate 2 memory blocks and store a string in each
    p1 = malloc( 34 );
    p2 = malloc( 38 );
    strcpy_s( p1, 34, "p1 points to a Normal allocation block" );
    strcpy_s( p2, 38, "p2 points to a Client allocation block" );

    // Free both memory blocks
    free( p2 );
    free( p1 );

    // Set the debug-heap flag to no longer keep freed blocks in the
    // heap's linked list and turn on Debug type allocations (CLIENT)
    tmpDbgFlag = _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG);
    tmpDbgFlag |= _CRTDBG_ALLOC_MEM_DF;
    tmpDbgFlag &= ~_CRTDBG_DELAY_FREE_MEM_DF;
    _CrtSetDbgFlag(tmpDbgFlag);

    // Explicitly call _malloc_dbg to obtain the filename and
    // line number of our allocation request and also so we can
    // allocate CLIENT type blocks specifically for tracking
    p1 = _malloc_dbg( 40, _NORMAL_BLOCK, __FILE__, __LINE__ );
    p2 = _malloc_dbg( 40, _CLIENT_BLOCK, __FILE__, __LINE__ );
    strcpy_s( p1, 40, "p1 points to a Normal allocation block" );
    strcpy_s( p2, 40, "p2 points to a Client allocation block" );

    // _free_dbg must be called to free the CLIENT block
    _free_dbg( p2, _CLIENT_BLOCK );
    free( p1 );

    // Allocate p1 again and then exit - this will leave unfreed
    // memory on the heap
    p1 = malloc( 10 );
}
```

See also

Debug Routines
_crtDbgFlag
_CrtCheckMemory

_CrtSetDebugFillThreshold

10/31/2018 • 2 minutes to read • Edit Online

Retrieves or modifies the threshold controlling buffer-filling behavior in debug functions.

Syntax

```
size_t _CrtSetDebugFillThreshold( size_t newThreshold );
```

Parameters

newThreshold
New threshold size in bytes.

Return value

The previous threshold value.

Remarks

The debug versions of some security-enhanced CRT functions fill the buffer passed to them with a special character (0xFE). This helps to find cases where the incorrect size was passed to the function. Unfortunately, it also reduces performance. To improve performance, use **_CrtSetDebugFillThreshold** to disable buffer-filling for buffers larger than the *newThreshold* threshold. A *newThreshold* value of 0 disables it for all buffers.

The default threshold is **SIZE_T_MAX**.

Here is a list of the affected functions.

- _ecvt_s
- _fcvt_s
- _gcvt_s
- _itoa_s, _ltoa_s, _ultoa_s, _i64toa_s, _ui64toa_s, _itow_s, _ltow_s, _ultow_s, _i64tow_s, _ui64tow_s
- _makepath_s, _wmakepath_s
- _mbsnbcat_s, _mbsnbcat_s_l
- _mbsnbcpy_s, _mbsnbcpy_s_l
- _mbsnbset_s, _mbsnbset_s_l
- _splitpath_s, _wsplitpath_s
- strcat_s, wcscat_s, _mbscat_s
- strcpy_s, wcscpy_s, _mbscpy_s
- strerror_s, _strerror_s, _wcserror_s, __wcserror_s
- _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l
- strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

- strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
- _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l
- _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l
- _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetDebugFillThreshold	<crtdbg.h>

This function is Microsoft-specific. For more compatibility information, see Compatibility.

Libraries

Debug versions of the C run-time libraries only.

Example

```
// crt_crtsetdebugfillthreshold.c
// compile with: cl /MTd crt_crtsetdebugfillthreshold.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>

void Clear( char buff[], size_t size )
{
   for( int i=0; i<size; ++i )
      buff[i] = 0;
}

void Print( char buff[], size_t size )
{
   for( int i=0; i<size; ++i )
      printf( "%02x  %c\n", (unsigned char)buff[i], buff[i] );
}

int main( void )
{
   char buff[10];

   printf( "With buffer-filling on:\n" );
   strcpy_s( buff, _countof(buff), "howdy" );
   Print( buff, _countof(buff) );

   _CrtSetDebugFillThreshold( 0 );

   printf( "With buffer-filling off:\n" );
   Clear( buff, _countof(buff) );
   strcpy_s( buff, _countof(buff), "howdy" );
   Print( buff, _countof(buff) );
}
```

```
With buffer-filling on:
68  h
6f  o
77  w
64  d
79  y
00
fe  ■
fe  ■
fe  ■
fe  ■
With buffer-filling off:
68  h
6f  o
77  w
64  d
79  y
00
00
00
00
00
```

See also

Debug Routines

_CrtSetDumpClient

10/31/2018 • 2 minutes to read • Edit Online

Installs an application-defined function to dump **_CLIENT_BLOCK** type memory blocks (debug version only).

Syntax

```
_CRT_DUMP_CLIENT _CrtSetDumpClient( _CRT_DUMP_CLIENT dumpClient );
```

Parameters

dumpClient
New client-defined memory dump function to hook into the C run-time debug memory dump process.

Return Value

Returns the previously defined client block dump function.

Remarks

The **_CrtSetDumpClient** function allows the application to hook its own function to dump objects stored in **_CLIENT_BLOCK** memory blocks into the C run-time debug memory dump process. As a result, every time a debug dump function such as _CrtMemDumpAllObjectsSince or _CrtDumpMemoryLeaks dumps a **_CLIENT_BLOCK** memory block, the application's dump function is called as well. **_CrtSetDumpClient** provides an application with an easy method for detecting memory leaks and validating or reporting the contents of data stored in **_CLIENT_BLOCK** blocks. When _DEBUG is not defined, calls to **_CrtSetDumpClient** are removed during preprocessing.

The **_CrtSetDumpClient** function installs the new application-defined dump function specified in *dumpClient* and returns the previously defined dump function. An example of a client block dump function is as follows:

```
void DumpClientFunction( void *userPortion, size_t blockSize );
```

The *userPortion* argument is a pointer to the beginning of the user data portion of the memory block and *blockSize* specifies the size of the allocated memory block in bytes. The client block dump function must return **void**. The pointer to the client dump function that is passed to **_CrtSetDumpClient** is of type **_CRT_DUMP_CLIENT**, as defined in Crtdbg.h:

```
typedef void (__cdecl *_CRT_DUMP_CLIENT)( void *, size_t );
```

For more information about functions that operate on **_CLIENT_BLOCK** type memory blocks, see Client Block Hook Functions. The _CrtReportBlockType function can be used to return information about block types and subtypes.

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetDumpClient	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines
_CrtReportBlockType
_CrtGetDumpClient

_CrtSetReportFile

10/31/2018 • 2 minutes to read • Edit Online

After you use _CrtSetReportMode to specify **_CRTDBG_MODE_FILE**, you can specify the file handle to receive the message text. **_CrtSetReportFile** is also used by _CrtDbgReport, _CrtDbgReportW to specify the destination of text (debug version only).

Syntax

```
_HFILE _CrtSetReportFile(
   int reportType,
   _HFILE reportFile
);
```

Parameters

reportType
Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

reportFile
New report file for *reportType*.

Return Value

On successful completion, **_CrtSetReportFile** returns the previous report file defined for the report type specified in *reportType*. If an invalid value is passed in for *reportType*, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_CRTDBG_HFILE_ERROR**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

_CrtSetReportFile is used with the _CrtSetReportMode function to define the destination or destinations for a specific report type generated by **_CrtDbgReport**. When **_CrtSetReportMode** has been called to assign the **_CRTDBG_MODE_FILE** reporting mode for a specific report type, **_CrtSetReportFile** should then be called to define the specific file or stream to use as the destination. When _DEBUG is not defined, calls to **_CrtSetReportFile** are removed during preprocessing.

The following list shows the available choices for *reportFile* and the resulting behavior of **_CrtDbgReport**. These options are defined as bit flags in Crtdbg.h.

- **file handle**

  A handle to the file that will be the destination for messages. No attempt is made to verify the validity of the handle. You must open and close the handle to the file. For example:

```
HANDLE hLogFile;
hLogFile = CreateFile("c:\\log.txt", GENERIC_WRITE,
    FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);
_CrtSetReportMode(_CRT_WARN, _CRTDBG_MODE_FILE);
_CrtSetReportFile(_CRT_WARN, hLogFile);

_RPT0(_CRT_WARN,"file message\n");
CloseHandle(hLogFile);
```

- **_CRTDBG_FILE_STDERR**

  Writes message to **stderr**, which can be redirected as follows:

```
freopen( "c:\\log2.txt", "w", stderr);
_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_FILE);
_CrtSetReportFile(_CRT_ERROR, _CRTDBG_FILE_STDERR);

_RPT0(_CRT_ERROR,"1st message\n");
```

- **_CRTDBG_FILE_STDOUT**

  Writes message to **stdout**, which you can redirect.

- **_CRTDBG_REPORT_FILE**

  Returns the current report mode.

The report file used by each report type can be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_ERROR** be reported to **stderr**, while a *reportType* of **_CRT_ASSERT** be reported to a user-defined file handle or stream.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_CrtSetReportFile	<crtdbg.h>	<errno.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For more compatibility information, see Compatibility.

Libraries: Debug versions of CRT Library Features only.

See also

Debug Routines

_CrtSetReportHook

10/31/2018 • 2 minutes to read • Edit Online

Installs a client-defined reporting function by hooking it into the C run-time debug reporting process (debug version only).

Syntax

```
_CRT_REPORT_HOOK _CrtSetReportHook(
   _CRT_REPORT_HOOK reportHook
);
```

Parameters

reportHook
New client-defined reporting function to hook into the C run-time debug reporting process.

Return Value

Returns the previous client-defined reporting function.

Remarks

_CrtSetReportHook allows an application to use its own reporting function into the C run-time debug library reporting process. As a result, whenever _CrtDbgReport is called to generate a debug report, the application's reporting function is called first. This functionality enables an application to perform operations such as filtering debug reports so it can focus on specific allocation types or send a report to destinations not available by using **_CrtDbgReport**. When _DEBUG is not defined, calls to **_CrtSetReportHook** are removed during preprocessing.

For a more robust version of **_CrtSetReportHook**, see _CrtSetReportHook2.

The **_CrtSetReportHook** function installs the new client-defined reporting function specified in *reportHook* and returns the previous client-defined hook. The following example demonstrates how a client-defined report hook should be prototyped:

```
int YourReportHook( int reportType, char *message, int *returnValue );
```

where *reportType* is the debug report type (**_CRT_WARN**, **_CRT_ERROR**, or **_CRT_ASSERT**), *message* is the fully assembled debug user message to be contained in the report, and **returnValue** is the value specified by the client-defined reporting function that should be returned by **_CrtDbgReport**. For a complete description of the available report types, see the _CrtSetReportMode function.

If the client-defined reporting function completely handles the debug message such that no further reporting is required, then the function should return **TRUE**. When the function returns **FALSE**, **_CrtDbgReport** is called to generate the debug report using the current settings for the report type, mode, and file. In addition, by specifying the **_CrtDbgReport** return value in **returnValue**, the application can also control whether a debug break occurs. For a complete description of how the debug report is configured and generated, see **_CrtSetReportMode**, _CrtSetReportFile, and **_CrtDbgReport**.

For more information about using other hook-capable run-time functions and writing your own client-defined hook functions, see Debug Hook Function Writing.

> **NOTE**
>
> If your application is compiled with **/clr** and the reporting function is called after the application has exited main, the CLR will throw an exception if the reporting function calls any CRT functions.

Requirements

ROUTINE	REQUIRED HEADER
_CrtSetReportHook	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines
_CrtGetReportHook

_CrtSetReportHook2, _CrtSetReportHookW2

10/31/2018 • 3 minutes to read • Edit Online

Installs or uninstalls a client-defined reporting function by hooking it into the C run-time debug reporting process (debug version only).

Syntax

```
int _CrtSetReportHook2(
   int mode,
   _CRT_REPORT_HOOK pfnNewHook
);
int _CrtSetReportHookW2(
   int mode,
   _CRT_REPORT_HOOKW pfnNewHook
);
```

Parameters

mode
The action to take: **_CRT_RPTHOOK_INSTALL** or **_CRT_RPTHOOK_REMOVE**.

pfnNewHook
Report hook to install or remove in the narrow-character or wide-character version of this function.

Return Value

-1 if an error was encountered, with **EINVAL** or **ENOMEM** set; otherwise returns the reference count of *pfnNewHook* after the call.

Remarks

_CrtSetReportHook2 and **_CrtSetReportHookW2** let you hook or unhook a function, whereas _CrtSetReportHook only lets you hook a function.

_CrtSetReportHook2 or **_CrtSetReportHookW2** should be used instead of **_CrtSetReportHook** when the hook call is made in a DLL and when multiple DLLs might be loaded and setting their own hook functions. In such a situation, DLLs can be unloaded in a different order than they were loaded and the hook function can be left pointing at an unloaded DLL. Any debug output crashes the process if the hook functions were added with **_CrtSetReportHook**.

Any hook functions added with **_CrtSetReportHook** are called if there are no hook functions added with **_CrtSetReportHook2** or **_CrtSetReportHookW2** or if all hook functions added with **_CrtSetReportHook2** and **_CrtSetReportHookW2** return **FALSE**.

The wide-character version of this function is available. The report hook functions take a string whose type (wide or narrow characters) must match the version of this function used. Use the following function prototype for the report hooks used with the wide-character version of this function:

```
int YourReportHook( int reportType, wchar_t *message, int *returnValue );
```

Use the following prototype for the narrow-character report hooks:

```
int YourReportHook( int reportType, char *message, int *returnValue );
```

These functions validate their parameters. If *mode* or **pfnNewNook** is invalid, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

> **NOTE**
> If your application is compiled with **/clr** and the reporting function is called after the application has exited main, the CLR will throw an exception if the reporting function calls any CRT functions.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_CrtSetReportHook2	<crtdbg.h>	<errno.h>
_CrtSetReportHookW2	<crtdbg.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_setreporthook2.c
#include <windows.h>
#include <stdio.h>
#include <crtdbg.h>
#include <assert.h>

int __cdecl TestHook1(int nReportType, char* szMsg, int* pnRet)
{
   int nRet = FALSE;

   printf("CRT report hook 1.\n");
   printf("CRT report type is \"");
   switch (nReportType)
   {
      case _CRT_ASSERT:
      {
         printf("_CRT_ASSERT");
         // nRet = TRUE;   // Always stop for this type of report
         break;
      }

      case _CRT_WARN:
      {
         printf("_CRT_WARN");
         break;
      }

      case _CRT_ERROR:
      {
         printf("_CRT_ERROR");
         break;
```

```
      }

      default:
      {
         printf("???Unknown???");
         break;
      }
   }

   printf("\".\nCRT report message is:\n\t");
   printf(szMsg);

   if   (pnRet)
      *pnRet = 0;

   return   nRet;
}

int __cdecl   TestHook2(int nReportType, char* szMsg, int* pnRet)
{
   int   nRet = FALSE;

   printf("CRT report hook 2.\n");
   printf("CRT report type is \"");
   switch   (nReportType)
   {
      case _CRT_WARN:
      {
         printf("_CRT_WARN");
         break;
      }

      case _CRT_ERROR:
      {
         printf("_CRT_ERROR");
         break;
      }

      case _CRT_ASSERT:
      {
         printf("_CRT_ASSERT");
         nRet = TRUE;   // Always stop for this type of report
         break;
      }

      default:
      {
         printf("???Unknown???");
         break;
      }
   }

   printf("\".\nCRT report message is: \t");
   printf(szMsg);

   if   (pnRet)
      *pnRet = 0;
   // printf("CRT report code is %d.\n", *pnRet);
   return   nRet;
}

int   main(int argc, char* argv[])
{
   int   nRet = 0;

   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1);
   printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1)"
          " returned %d\n", nRet);
   nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2);
```

```
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1)"
           " returned %d\n", nRet);

    _ASSERT(0);

    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2)"
           " returned %d\n", nRet);
    nRet = _CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1);
    printf("_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1)"
           " returned %d\n", nRet);

    return   nRet;
}
```

Output

```
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_INSTALL, TestHook1) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook2) returned 0
_CrtSetReportHook2(_CRT_RPTHOOK_REMOVE, TestHook1) returned 0
```

See also

Debug Routines

_CrtSetReportMode

10/31/2018 • 2 minutes to read • Edit Online

Specifies the destination or destinations for a specific report type generated by **_CrtDbgReport** and any macros that call _CrtDbgReport, _CrtDbgReportW, such as _ASSERT, _ASSERTE, _ASSERT_EXPR Macros, _ASSERT, _ASSERTE, _ASSERT_EXPR Macros, _RPT, _RPTF, _RPTW, _RPTFW Macros, and _RPT, _RPTF, _RPTW, _RPTFW Macros (debug version only).

Syntax

```
int _CrtSetReportMode(
   int reportType,
   int reportMode
);
```

Parameters

reportType
Report type: **_CRT_WARN**, **_CRT_ERROR**, and **_CRT_ASSERT**.

reportMode
New report mode or modes for *reportType*.

Return Value

On successful completion, **_CrtSetReportMode** returns the previous report mode or modes for the report type specified in *reportType*. If an invalid value is passed in as *reportType* or an invalid mode is specified for *reportMode*, **_CrtSetReportMode** invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

_CrtSetReportMode specifies the output destination for **_CrtDbgReport**. Because the macros _ASSERT, _ASSERTE, _RPT, and _RPTF call **_CrtDbgReport**, **_CrtSetReportMode** specifies the output destination of text specified with those macros.

When _DEBUG is not defined, calls to **_CrtSetReportMode** are removed during preprocessing.

If you do not call **_CrtSetReportMode** to define the output destination of messages, then the following defaults are in effect:

- Assertion failures and errors are directed to a debug message window.
- Warnings from Windows applications are sent to the debugger's output window.
- Warnings from console applications are not displayed.

The following table lists the report types defined in Crtdbg.h.

REPORT TYPE	DESCRIPTION
_CRT_WARN	Warnings, messages, and information that does not need immediate attention.
_CRT_ERROR	Errors, unrecoverable problems, and issues that require immediate attention.
_CRT_ASSERT	Assertion failures (asserted expressions that evaluate to **FALSE**).

The **_CrtSetReportMode** function assigns the new report mode specified in *reportMode* to the report type specified in *reportType* and returns the previously defined report mode for *reportType*. The following table lists the available choices for *reportMode* and the resulting behavior of **_CrtDbgReport**. These options are defined as bit flags in Crtdbg.h.

REPORT MODE	_CRTDBGREPORT BEHAVIOR
_CRTDBG_MODE_DEBUG	Writes the message to the debugger's output window.
_CRTDBG_MODE_FILE	Writes the message to a user-supplied file handle. _CrtSetReportFile should be called to define the specific file or stream to use as the destination.
_CRTDBG_MODE_WNDW	Creates a message box to display the message along with the abort, **Retry**, and **Ignore** buttons.
_CRTDBG_REPORT_MODE	Returns *reportMode* for the specified *reportType*: 1 **_CRTDBG_MODE_FILE** 2 **_CRTDBG_MODE_DEBUG** 4 **_CRTDBG_MODE_WNDW**

Each report type can be reported using one, two, or three modes or no mode at all. Therefore, it is possible to have more than one destination defined for a single report type. For example, the following code fragment causes assertion failures to be sent to both a debug message window and to **stderr**:

```
_CrtSetReportMode( _CRT_ASSERT, _CRTDBG_MODE_FILE | _CRTDBG_MODE_WNDW );
_CrtSetReportFile( _CRT_ASSERT, _CRTDBG_FILE_STDERR );
```

In addition, the reporting mode or modes for each report type can be separately controlled. For example, it is possible to specify that a *reportType* of **_CRT_WARN** be sent to an output debug string, while **_CRT_ASSERT** be displayed using a debug message window and sent to **stderr**, as previously illustrated.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_CrtSetReportMode	<crtdbg.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries: Debug versions of CRT Library Features only.

See also

Debug Routines

cscanf

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _cscanf instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data from the console. More secure versions of these functions are available; see _cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cscanf(
   const char *format [,
   argument] ...
);
int _cscanf_l(
   const char *format,
   locale_t locale [,
   argument] ...
);
int _cwscanf(
   const wchar_t *format [,
   argument] ...
);
int _cwscanf_l(
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional parameters.

locale
The locale to use.

Return Value

The number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is **EOF** for an attempt to read at end of file. This can occur when keyboard input is redirected at the operating-system command-line level. A return value of 0 means that no fields were assigned.

Remarks

The **_cscanf** function reads data directly from the console into the locations given by *argument*. The _getche

function is used to read characters. Each optional parameter must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* parameter for the scanf function. While **_cscanf** normally echoes the input character, it does not do so if the last call was to **_ungetch**.

This function validates its parameters. If format is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EOF**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscanf	**_cscanf**	**_cscanf**	**_cwscanf**
_tcscanf_l	**_cscanf_l**	**_cscanf_l**	**_cwscanf_l**

Requirements

ROUTINE	REQUIRED HEADER
_cscanf, **_cscanf_l**	<conio.h>
_cwscanf, **_cwscanf_l**	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cscanf.c
// compile with: /c /W3
/* This program prompts for a string
* and uses _cscanf to read in the response.
* Then _cscanf returns the number of items
* matched, and the program displays that number.
*/

#include <stdio.h>
#include <conio.h>

int main( void )
{
   int   result, i[3];

   _cprintf_s( "Enter three integers: ");
   result = _cscanf( "%i %i %i", &i[0], &i[1], &i[2] ); // C4996
   // Note: _cscanf is deprecated; consider using _cscanf_s instead
   _cprintf_s( "\r\nYou entered " );
   while( result-- )
      _cprintf_s( "%i ", i[result] );
   _cprintf_s( "\r\n" );
}
```

```
1 2 3
```

```
Enter three integers: 1 2 3
You entered 3 2 1
```

See also

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf, _sscanf_l, swscanf, _swscanf_l

_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data from the console. These more secure versions of _cscanf, _cscanf_l, _cwscanf, _cwscanf_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _cscanf_s(
   const char *format [,
   argument] ...
);
int _cscanf_s_l(
   const char *format,
   locale_t locale [,
   argument] ...
);
int _cwscanf_s(
   const wchar_t *format [,
   argument] ...
);
int _cwscanf_s_l(
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional parameters.

locale
The locale to use.

Return Value

The number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is **EOF** for an attempt to read at end of file. This can occur when keyboard input is redirected at the operating-system command-line level. A return value of 0 means that no fields were assigned.

These functions validate their parameters. If *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and **errno** is set to **EINVAL**.

Remarks

The **_cscanf_s** function reads data directly from the console into the locations given by *argument*. The _getche function is used to read characters. Each optional parameter must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* parameter for the scanf_s function. While **_cscanf_s** normally echoes the input character, it does not do so if the last call was to **_ungetch**.

Like other secure versions of functions in the **scanf** family, **_cscanf_s** and **_cswscanf_s** require size arguments for the type field characters **c**, **C**, **s**, **S**, and **[**. For more information, see scanf Width Specification.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscanf_s	**_cscanf_s**	**_cscanf_s**	**_cwscanf_s**
_tcscanf_s_l	**_cscanf_s_l**	**_cscanf_s_l**	**_cwscanf_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_cscanf_s, **_cscanf_s_l**	<conio.h>
_cwscanf_s, **_cwscanf_s_l**	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_cscanf_s.c
// compile with: /c
/* This program prompts for a string
* and uses _cscanf_s to read in the response.
* Then _cscanf_s returns the number of items
* matched, and the program displays that number.
*/

#include <stdio.h>
#include <conio.h>

int main( void )
{
   int result, n[3];
   int i;

   result = _cscanf_s( "%i %i %i", &n[0], &n[1], &n[2] );
   _cprintf_s( "\r\nYou entered " );
   for( i=0; i<result; i++ )
      _cprintf_s( "%i ", n[i] );
   _cprintf_s( "\r\n" );
}
```

```
1 2 3
```

```
You entered 1 2 3
```

See also

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

csin, csinf, csinl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the sine of a complex number.

Syntax

```
_Dcomplex csin(
   _Dcomplex z
);
_Fcomplex csin(
   _Fcomplex z
);  // C++ only
_Lcomplex csin(
   _Lcomplex z
);  // C++ only
_Fcomplex csinf(
   _Fcomplex z
);
_Lcomplex csinl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The sine of *z*, in radians.

Remarks

Because C++ allows overloading, you can call overloads of **csin** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **csin** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
csin, **csinf**, **csinl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl
csinh, csinhf, csinhl

casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

csinh, csinhf, csinhl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the hyperbolic sine of a complex number.

Syntax

```
_Dcomplex csinh(
   _Dcomplex z
);
_Fcomplex csinh(
   _Fcomplex z
);  // C++ only
_Lcomplex csinh(
   _Lcomplex z
);  // C++ only
_Fcomplex csinhf(
   _Fcomplex z
);
_Lcomplex csinhl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The hyperbolic sine of *z*, in radians.

Remarks

Because C++ allows overloading, you can call overloads of **csinh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **csinh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
csinh, **csinhf**, **csinhl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
catanh, catanhf, catanhl
ctanh, ctanhf, ctanhl
catan, catanf, catanl
casinh, casinhf, casinhl

ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

csqrt, csqrtf, csqrtl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the square root of a complex number, with a branch cut along the negative real axis.

Syntax

```
_Dcomplex csqrt(
   _Dcomplex z
);
_Fcomplex csqrt(
   _Fcomplex z
);  // C++ only
_Lcomplex csqrt(
   _Lcomplex z
);  // C++ only
_Fcomplex csqrtf(
   _Fcomplex z
);
_Lcomplex csqrtl(
   _Lcomplex z
);
```

Parameters

z

A complex number.

Return Value

The square root of *z*. The result is in the right half-plane.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± QNAN, IND	none	_DOMAIN
- ∞	none	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **csqrt** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **csqrt** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
csqrt, **csqrtf**, **csqrtl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

ctan, ctanf, ctanl

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the tangent of a complex number.

Syntax

```
_Dcomplex ctan(
   _Dcomplex z
);
_Fcomplex ctan(
   _Fcomplex z
);  // C++ only
_Lcomplex ctan(
   _Lcomplex z
);  // C++ only
_Fcomplex ctanf(
   _Fcomplex z
);
_Lcomplex ctanl(
   _Lcomplex z
);
```

Parameters

z

A complex number that represents the angle, in radians.

Return Value

The tangent of *z*.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± ∞, QNAN, IND	none	_DOMAIN
± ∞ (**tan**, **tanf**)	INVALID	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **ctan** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **ctan** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
ctan, **ctanf**, **ctanl**	<complex.h>	<ccomplex>

For more compatibility information, see Compatibility.

See also

ctanh, ctanhf, ctanhl

2/4/2019 • 2 minutes to read • Edit Online

Computes the complex hyperbolic tangent of a complex number.

Syntax

```
_Dcomplex ctanh(
   _Dcomplex z
);
_Fcomplex ctanh(
   _Fcomplex z
);  // C++ only
_Lcomplex ctanh(
   _Lcomplex z
);  // C++ only
_Fcomplex ctanhf(
   _Fcomplex z
);
_Lcomplex ctanhl(
   _Lcomplex z
);
```

Parameters

z
A complex number that represents an angle, in radians.

Return Value

The complex hyperbolic tangent of *z*.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± ∞, QNAN, IND	none	_DOMAIN
± ∞ (tan, tanf)	INVALID	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **ctanh** that take and return **_Fcomplex** and **_Lcomplex** values. In a C program, **ctanh** always takes and returns a **_Dcomplex** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
ctanh, **ctanhf**, **ctanhl**	<complex.h>	<ccomplex>

For compatibility information, see Compatibility.

See also

catanh, catanhf, catanhl
catan, catanf, catanl
csinh, csinhf, csinhl
casinh, casinhf, casinhl
ccosh, ccoshf, ccoshl
cacosh, cacoshf, cacoshl
cacos, cacosf, cacosl
ctan, ctanf, ctanl
csin, csinf, csinl
casin, casinf, casinl
ccos, ccosf, ccosl
csqrt, csqrtf, csqrtl

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64

10/31/2018 • 2 minutes to read • Edit Online

Convert a time value to a string and adjust for local time zone settings. More secure versions of these functions are available; see ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s.

Syntax

```
char *ctime( const time_t *sourceTime );
char *_ctime32( const __time32_t *sourceTime );
char *_ctime64( const __time64_t *sourceTime );
wchar_t *_wctime( const time_t *sourceTime );
wchar_t *_wctime32( const __time32_t *sourceTime );
wchar_t *_wctime64( const __time64_t *sourceTime );
```

Parameters

sourceTime
Pointer to stored time to convert.

Return Value

A pointer to the character string result. **NULL** will be returned if:

- *sourceTime* represents a date before midnight, January 1, 1970, UTC.
- If you use **_ctime32** or **_wctime32** and *sourceTime* represents a date after 23:59:59 January 18, 2038, UTC.
- If you use **_ctime64** or **_wctime64** and *sourceTime* represents a date after 23:59:59, December 31, 3000, UTC.

ctime is an inline function which evaluates to **_ctime64** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **ctime** to evaluate to **_ctime32**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

Remarks

The **ctime** function converts a time value stored as a time_t value into a character string. The *sourceTime* value is usually obtained from a call to time, which returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC). The return value string contains exactly 26 characters and has the form:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character ('\n') and the null character ('\0') occupy the last two positions of the string.

The converted character string is also adjusted according to the local time zone settings. See the time, _ftime,

and localtime functions for information on configuring the local time and the _tzset function for details about defining the time zone environment and global variables.

A call to **ctime** modifies the single statically allocated buffer used by the **gmtime** and **localtime** functions. Each call to one of these routines destroys the result of the previous call. **ctime** shares a static buffer with the **asctime** function. Thus, a call to **ctime** destroys the results of any previous call to **asctime**, **localtime**, or **gmtime**.

_wctime and **_wctime64** are the wide-character version of **ctime** and **_ctime64**; returning a pointer to wide-character string. Otherwise, **_ctime64**, **_wctime**, and **_wctime64** behave identically to **ctime**.

These functions validate their parameters. If *sourceTime* is a null pointer, or if the *sourceTime* value is negative, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tctime	**ctime**	**ctime**	**_wctime**
_tctime32	**_ctime32**	**_ctime32**	**_wctime32**
_tctime64	**_ctime64**	**_ctime64**	**_wctime64**

Requirements

ROUTINE	REQUIRED HEADER
ctime	<time.h>
_ctime32	<time.h>
_ctime64	<time.h>
_wctime	<time.h> or <wchar.h>
_wctime32	<time.h> or <wchar.h>
_wctime64	<time.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_ctime64.c
// compile with: /W3
/* This program gets the current
* time in _time64_t form, then uses ctime to
* display the time in string form.
*/

#include <time.h>
#include <stdio.h>

int main( void )
{
   __time64_t ltime;

   _time64( &ltime );
   printf( "The time is %s\n", _ctime64( &ltime ) ); // C4996
   // Note: _ctime64 is deprecated; consider using _ctime64_s
}
```

```
The time is Wed Feb 13 16:04:43 2002
```

See also

Time Management
asctime, _wasctime
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s

10/31/2018 • 3 minutes to read • Edit Online

Convert a time value to a string and adjust for local time zone settings. These are versions of ctime, _ctime64, _wctime, _wctime64 with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t ctime_s(
   char* buffer,
   size_t numberOfElements,
   const time_t *sourceTime
);
errno_t _ctime32_s(
   char* buffer,
   size_t numberOfElements,
   const __time32_t *sourceTime
);
errno_t _ctime64_s(
   char* buffer,
   size_t numberOfElements,
   const __time64_t *sourceTime )
;
errno_t _wctime_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const time_t *sourceTime
);
errno_t _wctime32_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const __time32_t *sourceTime
);
errno_t _wctime64_s(
   wchar_t* buffer,
   size_t numberOfElements,
   const __time64_t *sourceTime
);
```

```
template <size_t size>
errno_t _ctime32_s(
   char (&buffer)[size],
   const __time32_t *sourceTime
); // C++ only
template <size_t size>
errno_t _ctime64_s(
   char (&buffer)[size],
   const __time64_t *sourceTime
); // C++ only
template <size_t size>
errno_t _wctime32_s(
   wchar_t (&buffer)[size],
   const __time32_t *sourceTime
); // C++ only
template <size_t size>
errno_t _wctime64_s(
   wchar_t (&buffer)[size],
   const __time64_t *sourceTime
); // C++ only
```

Parameters

buffer
Must be large enough to hold 26 characters. A pointer to the character string result, or **NULL** if:

- *sourceTime* represents a date before midnight, January 1, 1970, UTC.
- If you use **_ctime32_s** or **_wctime32_s** and *sourceTime* represents a date after 23:59:59 January 18, 2038, UTC.
- If you use **_ctime64_s** or **_wctime64_s** and *sourceTime* represents a date after 23:59:59, December 31, 3000, UTC.
- If you use **_ctime_s** or **_wctime_s**, these functions are wrappers to the previous functions. See the Remarks section.

numberOfElements
The size of the buffer.

sourceTime
Pointer to stored time.

Return Value

Zero if successful. If there is a failure due to an invalid parameter, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, an error code is returned. Error codes are defined in ERRNO.H; for a listing of these errors, see errno. The actual error codes thrown for each error condition are shown in the following table.

Error Conditions

BUFFER	*NUMBEROFELEMENTS*	*SOURCETIME*	RETURN	VALUE IN *BUFFER*
NULL	any	any	**EINVAL**	Not modified
Not **NULL** (points to valid memory)	0	any	**EINVAL**	Not modified

BUFFER	*NUMBEROFELEMENTS*	*SOURCETIME*	RETURN	VALUE IN *BUFFER*
Not **NULL**	0< size < 26	any	**EINVAL**	Empty string
Not **NULL**	>= 26	NULL	**EINVAL**	Empty string
Not **NULL**	>= 26	< 0	**EINVAL**	Empty string

Remarks

The **ctime_s** function converts a time value stored as a time_t structure into a character string. The *sourceTime* value is usually obtained from a call to time, which returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC). The return value string contains exactly 26 characters and has the form:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The new line character ('\n') and the null character ('\0') occupy the last two positions of the string.

The converted character string is also adjusted according to the local time zone settings. See the time, _ftime, and localtime32_s functions for information about configuring the local time and the _tzset function for information about defining the time zone environment and global variables.

_wctime32_s and **_wctime64_s** are the wide-character version of **_ctime32_s** and **_ctime64_s**; returning a pointer to wide-character string. Otherwise, **_ctime64_s**, **_wctime32_s**, and **_wctime64_s** behave identically to **_ctime32_s**.

ctime_s is an inline function that evaluates to **_ctime64_s** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **ctime_s** to evaluate to **_ctime32_s**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tctime_s	**ctime_s**	**ctime_s**	**_wctime_s**
_tctime32_s	**_ctime32_s**	**_ctime32_s**	**_wctime32_s**
_tctime64_s	**_ctime64_s**	**_ctime64_s**	**_wctime64_s**

Requirements

ROUTINE	REQUIRED HEADER
ctime_s, **_ctime32_s**, **_ctime64_s**	<time.h>

ROUTINE	REQUIRED HEADER
_wctime_s, **_wctime32_s**, **_wctime64_s**	<time.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_wctime_s.c
// This program gets the current
// time in time_t form and then uses _wctime_s to
// display the time in string form.

#include <time.h>
#include <stdio.h>

#define SIZE 26

int main( void )
{
   time_t ltime;
   wchar_t buf[SIZE];
   errno_t err;

   time( &ltime );

   err = _wctime_s( buf, SIZE, &ltime );
   if (err != 0)
   {
      printf("Invalid Arguments for _wctime_s. Error Code: %d\n", err);
   }
   wprintf_s( L"The time is %s\n", buf );
}
```

```
The time is Fri Apr 25 13:03:39 2003
```

See also

Time Management
asctime_s, _wasctime_s
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64

cwait

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _cwait instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_cwait

10/31/2018 • 3 minutes to read • Edit Online

Waits until another process terminates.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _cwait(
   int *termstat,
   intptr_t procHandle,
   int action
);
```

Parameters

termstat
Pointer to a buffer where the result code of the specified process will be stored, or **NULL**.

procHandle
The handle to the process to wait on (that is, the process that has to terminate before **_cwait** can return).

action
NULL: Ignored by Windows operating system applications; for other applications: action code to perform on *procHandle*.

Return Value

When the specified process has successfully completed, returns the handle of the specified process and sets *termstat* to the result code that's returned by the specified process. Otherwise, returns -1 and sets **errno** as follows.

VALUE	DESCRIPTION
ECHILD	No specified process exists, *procHandle* is invalid, or the call to the GetExitCodeProcess or WaitForSingleObject API failed.
EINVAL	*action* is invalid.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_cwait** function waits for the termination of the process ID of the specified process that's provided by *procHandle*. The value of *procHandle* that's passed to **_cwait** should be the value that's returned by the call to the _spawn function that created the specified process. If the process ID terminates before **_cwait** is called, **_cwait** returns immediately. **_cwait** can be used by any process to wait for any other known process for which a valid

handle (*procHandle*) exists.

termstat points to a buffer where the return code of the specified process will be stored. The value of *termstat* indicates whether the specified process terminated normally by calling the Windows ExitProcess API. **ExitProcess** is called internally if the specified process calls **exit** or **_exit**, returns from **main**, or reaches the end of **main**. For more information about the value that's passed back through *termstat*, see GetExitCodeProcess. If **_cwait** is called by using a **NULL** value for *termstat*, the return code of the specified process is not stored.

The *action* parameter is ignored by the Windows operating system because parent-child relationships are not implemented in these environments.

Unless *procHandle* is -1 or -2 (handles to the current process or thread), the handle will be closed. Therefore, in this situation, do not use the returned handle.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_cwait	<process.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_cwait.c
// compile with: /c
// This program launches several processes and waits
// for a specified process to finish.

#define _CRT_RAND_S

#include <windows.h>
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Macro to get a random integer within a specified range
#define getrandom( min, max ) (( (rand_s (&number), number) % (int)((( max ) + 1 ) - ( min ))) + ( min ))

struct PROCESS
{
   int     nPid;
   char    name[40];
} process[4] = { { 0, "Ann" }, { 0, "Beth" }, { 0, "Carl" }, { 0, "Dave" } };

int main( int argc, char *argv[] )
{
   int termstat, c;
   unsigned int number;

   srand( (unsigned)time( NULL ) );    // Seed randomizer

   // If no arguments, this is the calling process
   if ( argc == 1 )
   {
      // Spawn processes in numeric order
      for ( c = 0; c < 4; c++ ) {
         _flushall();
         process[c].nPid = _spawnl( _P_NOWAIT, argv[0], argv[0],
                                    process[c].name, NULL );
      }

      // Wait for randomly specified process, and respond when done
      c = getrandom( 0, 3 );
      printf( "Come here, %s.\n", process[c].name );
      _cwait( &termstat, process[c].nPid, _WAIT_CHILD );
      printf( "Thank you, %s.\n", process[c].name );

   }
   // If there are arguments, this must be a spawned process
   else
   {
      // Delay for a period that's determined by process number
      Sleep( (argv[1][0] - 'A' + 1) * 1000L );
      printf( "Hi, Dad. It's %s.\n", argv[1] );
   }
}
```

```
Hi, Dad. It's Ann.
Come here, Ann.
Thank you, Ann.
Hi, Dad. It's Beth.
Hi, Dad. It's Carl.
Hi, Dad. It's Dave.
```

See also

_CxxThrowException

10/31/2018 • 2 minutes to read • Edit Online

Builds the exception record and calls the runtime environment to start processing the exception.

Syntax

```
extern "C" void __stdcall _CxxThrowException(
   void* pExceptionObject
   _ThrowInfo* pThrowInfo
);
```

Parameters

pExceptionObject
The object that generated the exception.

pThrowInfo
The information that is required to process the exception.

Remarks

This method is included in a compiler-only file that the compiler uses to process exceptions. Do not call the method directly from your code.

Requirements

Source: Throw.cpp

See also

Alphabetical Function Reference

difftime, _difftime32, _difftime64

11/8/2018 • 2 minutes to read • Edit Online

Finds the difference between two times.

Syntax

```
double difftime( time_t timeEnd, time_t timeStart );
double _difftime32( __time32_t timeEnd, __time32_t timeStart );
double _difftime64( __time64_t timeEnd, __time64_t timeStart );
```

Parameters

timeEnd
Ending time.

timeStart
Beginning time.

Return Value

difftime returns the elapsed time in seconds, from *timeStart* to *timeEnd*. The value returned is a double precision floating-point number. The return value may be 0, indicating an error.

Remarks

The **difftime** function computes the difference between the two supplied time values *timeStart* and *timeEnd*.

The time value supplied must fit within the range of **time_t**. **time_t** is a 64-bit value. Thus, the end of the range was extended from 23:59:59 January 18, 2038, UTC to 23:59:59, December 31, 3000. The lower range of **time_t** is still midnight, January 1, 1970.

difftime is an inline function that evaluates to either **_difftime32** or **_difftime64** depending on whether **_USE_32BIT_TIME_T** is defined. _difftime32 and _difftime64 can be used directly to force the use of a particular size of the time type.

These functions validate their parameters. If either of the parameters is zero or negative, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return 0 and set **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
difftime	<time.h>
_difftime32	<time.h>
_difftime64	<time.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_difftime.c
// This program calculates the amount of time
// needed to do a floating-point multiply 100 million times.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>

double RangedRand( float range_min, float range_max)
{
   // Generate random numbers in the half-closed interval
   // [range_min, range_max). In other words,
   // range_min <= random number < range_max
   return ((double)rand() / (RAND_MAX + 1) * (range_max - range_min)
            + range_min);
}

int main( void )
{
   time_t   start, finish;
   long     loop;
   double   result, elapsed_time;
   double   arNums[3];

   // Seed the random-number generator with the current time so that
   // the numbers will be different every time we run.
   srand( (unsigned)time( NULL ) );

   arNums[0] = RangedRand(1, FLT_MAX);
   arNums[1] = RangedRand(1, FLT_MAX);
   arNums[2] = RangedRand(1, FLT_MAX);
   printf( "Using floating point numbers %.5e %.5e %.5e\n", arNums[0], arNums[1], arNums[2] );

   printf( "Multiplying 2 numbers 100 million times...\n" );

   time( &start );
   for( loop = 0; loop < 100000000; loop++ )
      result = arNums[loop%3] * arNums[(loop+1)%3];
   time( &finish );

   elapsed_time = difftime( finish, start );
   printf( "\nProgram takes %6.0f seconds.\n", elapsed_time );
}
```

```
Using random floating point numbers 1.04749e+038 2.01482e+038 1.72737e+038
Multiplying 2 floating point numbers 100 million times...
Program takes      3 seconds.
```

See also

Floating-Point Support
Time Management
time, _time32, _time64

div, ldiv, lldiv

10/31/2018 • 2 minutes to read • Edit Online

Computes the quotient and the remainder of two integer values.

Syntax

```
div_t div(
   int numer,
   int denom
);
ldiv_t ldiv(
   long numer,
   long denom
);
lldiv_t lldiv(
   long long numer,
   long long denom
);
```

```
ldiv_t div(
   long numer,
   long denom
); /* C++ only */
lldiv_t div(
   long long numer,
   long long denom
); /* C++ only */
```

Parameters

numer
The numerator.

denom
The denominator.

Return Value

div called by using arguments of type **int** returns a structure of type **div_t**, which comprises the quotient and the remainder. The return value with arguments of type **long** is **ldiv_t**, and the return value with arguments of type **long long** is **lldiv_t**. **div_t**, **ldiv_t**, and **lldiv_t** are defined in <stdlib.h>.

Remarks

The **div** function divides *numer* by *denom* and thereby computes the quotient and the remainder. The div_t structure contains the quotient, **quot**, and the remainder, **rem**. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program terminates with an error message.

The overloads of **div** that take arguments of type **long** or **long long** are only available to C++ code. The return types ldiv_t and lldiv_t contains members **quot** and **rem**, which have the same meanings as the members of **div_t**.

Requirements

ROUTINE	REQUIRED HEADER
div, **ldiv**, **lldiv**	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_div.c
// arguments: 876 13

// This example takes two integers as command-line
// arguments and displays the results of the integer
// division. This program accepts two arguments on the
// command line following the program name, then calls
// div to divide the first argument by the second.
// Finally, it prints the structure members quot and rem.
//

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( int argc, char *argv[] )
{
   int x,y;
   div_t div_result;

   x = atoi( argv[1] );
   y = atoi( argv[2] );

   printf( "x is %d, y is %d\n", x, y );
   div_result = div( x, y );
   printf( "The quotient is %d, and the remainder is %d\n",
           div_result.quot, div_result.rem );
}
```

```
x is 876, y is 13
The quotient is 67, and the remainder is 5
```

See also

Floating-Point Support
ldiv, lldiv
imaxdiv

dup, dup2

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _dup, _dup2 instead.

_dup, _dup2

10/31/2018 • 2 minutes to read • Edit Online

Creates a second file descriptor for an open file (**_dup**), or reassigns a file descriptor (**_dup2**).

Syntax

```
int _dup( int fd );
int _dup2( int fd1, int fd2 );
```

Parameters

fd, *fd1*
File descriptors referring to open file.

fd2
Any file descriptor.

Return Value

_dup returns a new file descriptor. **_dup2** returns 0 to indicate success. If an error occurs, each function returns -1 and sets **errno** to **EBADF** if the file descriptor is invalid or to **EMFILE** if no more file descriptors are available. In the case of an invalid file descriptor, the function also invokes the invalid parameter handler, as described in Parameter Validation.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_dup** and **_dup2** functions associate a second file descriptor with a currently open file. These functions can be used to associate a predefined file descriptor, such as that for **stdout**, with a different file. Operations on the file can be carried out using either file descriptor. The type of access allowed for the file is unaffected by the creation of a new descriptor. **_dup** returns the next available file descriptor for the given file. **_dup2** forces *fd2* to refer to the same file as *fd1*. If *fd2* is associated with an open file at the time of the call, that file is closed.

Both **_dup** and **_dup2** accept file descriptors as parameters. To pass a stream (`FILE *`) to either of these functions, use _fileno. The **fileno** routine returns the file descriptor currently associated with the given stream. The following example shows how to associate **stderr** (defined as `FILE *` in Stdio.h) with a file descriptor:

```
int cstderr = _dup( _fileno( stderr ));
```

Requirements

ROUTINE	REQUIRED HEADER
_dup	<io.h>
_dup2	<io.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For more compatibility information, see Compatibility.

Example

```
// crt_dup.c
// This program uses the variable old to save
// the original stdout. It then opens a new file named
// DataFile and forces stdout to refer to it. Finally, it
// restores stdout to its original state.

#include <io.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int old;
   FILE *DataFile;

   old = _dup( 1 );   // "old" now refers to "stdout"
                      // Note:  file descriptor 1 == "stdout"
   if( old == -1 )
   {
      perror( "_dup( 1 ) failure" );
      exit( 1 );
   }
   _write( old, "This goes to stdout first\n", 26 );
   if( fopen_s( &DataFile, "data", "w" ) != 0 )
   {
      puts( "Can't open file 'data'\n" );
      exit( 1 );
   }

   // stdout now refers to file "data"
   if( -1 == _dup2( _fileno( DataFile ), 1 ) )
   {
      perror( "Can't _dup2 stdout" );
      exit( 1 );
   }
   puts( "This goes to file 'data'\n" );

   // Flush stdout stream buffer so it goes to correct file
   fflush( stdout );
   fclose( DataFile );

   // Restore original stdout
   _dup2( old, 1 );
   puts( "This goes to stdout\n" );
   puts( "The file 'data' contains:" );
   _flushall();
   system( "type data" );
}
```

```
This goes to stdout first
This goes to stdout

The file 'data' contains:
This goes to file 'data'
```

See also

Low-Level I/O
_close
_creat, _wcreat
_open, _wopen

_dupenv_s, _wdupenv_s

10/31/2018 • 2 minutes to read • Edit Online

Gets a value from the current environment.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _dupenv_s(
   char **buffer,
   size_t *numberOfElements,
   const char *varname
);
errno_t _wdupenv_s(
   wchar_t **buffer,
   size_t *numberOfElements,
   const wchar_t *varname
);
```

Parameters

buffer
Buffer to store the variable's value.

numberOfElements
Size of *buffer*.

varname
Environment variable name.

Return Value

Zero on success, an error code on failure.

These functions validate their parameters; if *buffer* or *varname* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

If these functions cannot allocate enough memory, they set *buffer* to **NULL** and *numberOfElements* to 0, and return **ENOMEM**.

Remarks

The **_dupenv_s** function searches the list of environment variables for *varname*. If the variable is found, **_dupenv_s** allocates a buffer and copies the variable's value into the buffer. The buffer's address and length are returned in *buffer* and *numberOfElements*. By allocating the buffer itself, **_dupenv_s** provides a more convenient alternative to getenv_s, _wgetenv_s.

> **NOTE**
> It is the calling program's responsibility to free the memory by calling free.

If the variable is not found, then *buffer* is set to **NULL**, *numberOfElements* is set to 0, and the return value is 0 because this situation is not considered to be an error condition.

If you are not interested in the size of the buffer you can pass **NULL** for *numberOfElements*.

_dupenv_s is not case sensitive in the Windows operating system. **_dupenv_s** uses the copy of the environment pointed to by the global variable **_environ** to access the environment. See the Remarks in getenv_s, _wgetenv_s for a discussion of **_environ**.

The value in *buffer* is a copy of the environment variable's value; modifying it has no effect on the environment. Use the _putenv_s, _wputenv_s function to modify the value of an environment variable.

_wdupenv_s is a wide-character version of **_dupenv_s**; the arguments of **_wdupenv_s** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**. See the Remarks in getenv_s, _wgetenv_s for more on **_wenviron**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tdupenv_s	**_dupenv_s**	**_dupenv_s**	**_wdupenv_s**

Requirements

ROUTINE	REQUIRED HEADER
_dupenv_s	<stdlib.h>
_wdupenv_s	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_dupenv_s.c
#include  <stdlib.h>

int main( void )
{
   char *pValue;
   size_t len;
   errno_t err = _dupenv_s( &pValue, &len, "pathext" );
   if ( err ) return -1;
   printf( "pathext = %s\n", pValue );
   free( pValue );
   err = _dupenv_s( &pValue, &len, "nonexistentvariable" );
   if ( err ) return -1;
   printf( "nonexistentvariable = %s\n", pValue );
   free( pValue ); // It's OK to call free with NULL
}
```

```
pathext = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.pl
nonexistentvariable = (null)
```

See also

Process and Environment Control
Environmental Constants
_dupenv_s_dbg, _wdupenv_s_dbg
getenv_s, _wgetenv_s
_putenv_s, _wputenv_s

_dupenv_s_dbg, _wdupenv_s_dbg

10/31/2018 • 2 minutes to read • Edit Online

Get a value from the current environment. Versions of _dupenv_s, _wdupenv_s that allocate memory with _malloc_dbg to provide additional debugging information.

Syntax

```
errno_t _dupenv_s_dbg(
   char **buffer,
   size_t *numberOfElements,
   const char *varname,
   int blockType,
   const char *filename,
   int linenumber
);
errno_t _wdupenv_s_dbg(
   wchar_t **buffer,
   size_t * numberOfElements,
   const wchar_t *varname,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

buffer
Buffer to store the variable's value.

numberOfElements
Size of *buffer*.

varname
Environment variable name.

blockType
Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file or **NULL**.

linenumber
Line number in source file or **NULL**.

Return Value

Zero on success, an error code on failure.

These functions validate their parameters; if *buffer* or *varname* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

If these functions cannot allocate enough memory, they set *buffer* to **NULL** and *numberOfElements* to 0, and return **ENOMEM**.

Remarks

The **_dupenv_s_dbg** and **_wdupenv_s_dbg** functions are identical to **_dupenv_s** and **_wdupenv_s** except that, when **_DEBUG** is defined, these functions use the debug version of malloc, _malloc_dbg, to allocate memory for the value of the environment variable. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_dupenv_s** and **_wdupenv_s** are remapped to **_dupenv_s_dbg** and **_wdupenv_s_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information on block types, see Types of blocks on the debug heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tdupenv_s_dbg	**_dupenv_s_dbg**	**_dupenv_s_dbg**	**_wdupenv_s_dbg**

Requirements

ROUTINE	REQUIRED HEADER
_dupenv_s_dbg	<crtdbg.h>
_wdupenv_s_dbg	<crtdbg.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_dupenv_s_dbg.c
#include  <stdlib.h>
#include <crtdbg.h>

int main( void )
{
   char *pValue;
   size_t len;
   errno_t err = _dupenv_s_dbg( &pValue, &len, "pathext",
      _NORMAL_BLOCK, __FILE__, __LINE__ );
   if ( err ) return -1;
   printf( "pathext = %s\n", pValue );
   free( pValue );
   err = _dupenv_s_dbg( &pValue, &len, "nonexistentvariable",
      _NORMAL_BLOCK, __FILE__, __LINE__ );
   if ( err ) return -1;
   printf( "nonexistentvariable = %s\n", pValue );
   free( pValue ); // It's OK to call free with NULL
}
```

```
pathext = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.pl
nonexistentvariable = (null)
```

See also

Process and Environment Control
Environmental Constants
getenv_s, _wgetenv_s
_putenv_s, _wputenv_s

ecvt

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _ecvt or security-enhanced _ecvt_s instead.

_ecvt

10/31/2018 • 2 minutes to read • Edit Online

Converts a **double** number to a string. A more secure version of this function is available; see _ecvt_s.

Syntax

```
char *_ecvt(
   double value,
   int count,
   int *dec,
   int *sign
);
```

Parameters

value
Number to be converted.

count
Number of digits stored.

dec
Stored decimal-point position.

sign
Sign of the converted number.

Return Value

_ecvt returns a pointer to the string of digits; **NULL** if an error occurred.

Remarks

The **_ecvt** function converts a floating-point number to a character string. The *value* parameter is the floating-point number to be converted. This function stores up to *count* digits of *value* as a string and appends a null character ('\0'). If the number of digits in *value* exceeds *count*, the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

The total number of digits returned by **_ecvt** will not exceed **_CVTBUFSIZE**.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The *sign* parameter points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

The difference between **_ecvt** and **_fcvt** is in the interpretation of the *count* parameter. **_ecvt** interprets *count* as the total number of digits in the output string, whereas **_fcvt** interprets *count* as the number of digits after the decimal point.

_ecvt and **_fcvt** use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

This function validates its parameters. If *dec* or *sign* is **NULL**, or *count* is 0, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and **NULL** is returned.

Requirements

FUNCTION	REQUIRED HEADER
_ecvt	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_ecvt.c
// compile with: /W3
// This program uses _ecvt to convert a
// floating-point number to a character string.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int     decimal,   sign;
   char    *buffer;
   int     precision = 10;
   double  source = 3.1415926535;

   buffer = _ecvt( source, precision, &decimal, &sign ); // C4996
   // Note: _ecvt is deprecated; consider using _ecvt_s instead
   printf( "source: %2.10f   buffer: '%s'  decimal: %d  sign: %d\n",
           source, buffer, decimal, sign );
}
```

```
source: 3.1415926535   buffer: '3141592654'  decimal: 1  sign: 0
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_fcvt
_gcvt

_ecvt_s

10/31/2018 • 3 minutes to read • Edit Online

Converts a **double** number to a string. This is a version of _ecvt with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _ecvt_s(
   char * _Buffer,
   size_t _SizeInBytes,
   double _Value,
   int _Count,
   int *_Dec,
   int *_Sign
);
template <size_t size>
errno_t _ecvt_s(
   char (&_Buffer)[size],
   double _Value,
   int _Count,
   int *_Dec,
   int *_Sign
); // C++ only
```

Parameters

_Buffer
Filled with the pointer to the string of digits, the result of the conversion.

_SizeInBytes
Size of the buffer in bytes.

_Value
Number to be converted.

_Count
Number of digits stored.

_Dec
Stored decimal-point position.

_Sign
Sign of the converted number.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

In the case of an invalid parameter, as listed in the following table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

_BUFFER	*_SIZEINBYTES*	_VALUE	_COUNT	_DEC	_SIGN	RETURN VALUE	VALUE IN *BUFFER*
NULL	any	any	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	<=0	any	any	any	any	**EINVAL**	Not modified.
any	any	any	any	**NULL**	any	**EINVAL**	Not modified.
any	any	any	any	any	**NULL**	**EINVAL**	Not modified.

Security Issues

_ecvt_s might generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_ecvt_s** function converts a floating-point number to a character string. The *_Value* parameter is the floating-point number to be converted. This function stores up to *count* digits of *_Value* as a string and appends a null character ('\0'). If the number of digits in *_Value* exceeds *_Count*, the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *_Value* can be obtained from *_Dec* and *_Sign* after the call. The *_Dec* parameter points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The *_Sign* parameter points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

A buffer of length **_CVTBUFSIZE** is sufficient for any floating-point value.

The difference between **_ecvt_s** and **_fcvt_s** is in the interpretation of the *_Count* parameter. **_ecvt_s** interprets *_Count* as the total number of digits in the output string, whereas **_fcvt_s** interprets *_Count* as the number of digits after the decimal point.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_ecvt_s	<stdlib.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// ecvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main( )
{
    char * buf = 0;
    int decimal;
    int sign;
    int err;

    buf = (char*) malloc(_CVTBUFSIZE);
    err = _ecvt_s(buf, _CVTBUFSIZE, 1.2, 5, &decimal, &sign);

    if (err != 0)
    {
        printf("_ecvt_s failed with error code %d\n", err);
        exit(1);
    }

    printf("Converted value: %s\n", buf);
}
```

```
Converted value: 12000
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt
_fcvt_s
_gcvt_s

_endthread, _endthreadex

1/24/2019 • 2 minutes to read • Edit Online

Terminates a thread; **_endthread** terminates a thread that's created by **_beginthread** and **_endthreadex** terminates a thread that's created by **_beginthreadex**.

Syntax

```
void _endthread( void );
void _endthreadex(
   unsigned retval
);
```

Parameters

retval
Thread exit code.

Remarks

You can call **_endthread** or **_endthreadex** explicitly to terminate a thread; however, **_endthread** or **_endthreadex** is called automatically when the thread returns from the routine passed as a parameter to **_beginthread** or **_beginthreadex**. Terminating a thread with a call to **endthread** or **_endthreadex** helps ensure proper recovery of resources allocated for the thread.

> **NOTE**
> For an executable file linked with Libcmt.lib, do not call the Win32 ExitThread API; this prevents the run-time system from reclaiming allocated resources. **_endthread** and **_endthreadex** reclaim allocated thread resources and then call **ExitThread**.

_endthread automatically closes the thread handle. (This behavior differs from the Win32 **ExitThread** API.) Therefore, when you use **_beginthread** and **_endthread**, do not explicitly close the thread handle by calling the Win32 CloseHandle API.

Like the Win32 **ExitThread** API, **_endthreadex** does not close the thread handle. Therefore, when you use **_beginthreadex** and **_endthreadex**, you must close the thread handle by calling the Win32 **CloseHandle** API.

> **NOTE**
> **_endthread** and **_endthreadex** cause C++ destructors pending in the thread not to be called.

Requirements

FUNCTION	REQUIRED HEADER
_endthread	<process.h>
_endthreadex	<process.h>

For more compatibility information, see Compatibility.

Libraries

Multithreaded versions of the C run-time libraries only.

Example

See the example for _beginthread.

See also

Process and Environment Control
_beginthread, _beginthreadex

eof

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _eof instead.

_eof

10/31/2018 • 2 minutes to read • Edit Online

Tests for end of file (EOF).

Syntax

```
int _eof(
   int fd
);
```

Parameters

fd
File descriptor referring to the open file.

Return Value

_eof returns 1 if the current position is end of file, or 0 if it is not. A return value of -1 indicates an error; in this case, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EBADF**, which indicates an invalid file descriptor.

Remarks

The **_eof** function determines whether the end of the file associated with *fd* has been reached.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_eof	<io.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_eof.c
// This program reads data from a file
// ten bytes at a time until the end of the
// file is reached or an error is encountered.
//
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <share.h>

int main( void )
{
   int  fh, count, total = 0;
   char buf[10];
   if( _sopen_s( &fh, "crt_eof.txt", _O_RDONLY, _SH_DENYNO, 0 ) )
   {
        perror( "Open failed");
        exit( 1 );
   }
   // Cycle until end of file reached:
   while( !_eof( fh ) )
   {
      // Attempt to read in 10 bytes:
      if( (count = _read( fh, buf, 10 )) == -1 )
      {
         perror( "Read error" );
         break;
      }
      // Total actual bytes read
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   _close( fh );
}
```

Input: crt_eof.txt

```
This file contains some text.
```

Output

```
Number of bytes read = 29
```

See also

Error Handling
Low-Level I/O
clearerr
feof
ferror
perror, _wperror

erf, erff, erfl, erfc, erfcf, erfcl

2/4/2019 • 2 minutes to read • Edit Online

Computes the error function or the complementary error function of a value.

Syntax

```
double erf(
   double x
);
float erf(
   float x
); // C++ only
long double erf(
   long double x
); // C++ only
float erff(
   float x
);
long double erfl(
   long double x
);
double erfc(
   double x
);
float erfc(
   float x
); // C++ only
long double erfc(
   long double x
); // C++ only
float erfcf(
   float x
);
long double erfcl(
   long double x
);
```

Parameters

x
A floating-point value.

Return Value

The **erf** functions return the Gauss error function of *x*. The **erfc** functions return the complementary Gauss error function of *x*.

Remarks

The **erf** functions calculate the Gauss error function of *x*, which is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

The complementary Gauss error function is defined as 1 - erf(x). The **erf** functions return a value in the range -

1.0 to 1.0. There is no error return. The **erfc** functions return a value in the range 0 to 2. If *x* is too large for **erfc**, the **errno** variable is set to **ERANGE**.

Because C++ allows overloading, you can call overloads of **erf** and **erfc** that take and return **float** and **long double** types. In a C program, **erf** and **erfc** always take and return a **double**.

Requirements

FUNCTION	REQUIRED HEADER
erf, **erff**, **erfl**, **erfc**, **erfcf**, **erfcl**	<math.h>

For additional compatibility information, see Compatibility.

See also

Floating-Point Support

execl

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execl instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execl, _wexecl

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execl(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL
);
intptr_t _wexecl(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

cmdname
Path of the file to be executed.

arg0, ... *argn*
List of pointers to the parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter (one or more of the parameters was a null pointer or empty string).
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path is not found.

ERRNO VALUE	DESCRIPTION
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter. The first argument is the command or executable file name, and the second argument should be the same as the first. It becomes `argv[0]` in the executed process. The third argument is the first argument, `argv[1]`, of the process being executed.

The **_execl** functions validate their parameters. If either *cmdname* or *arg0* is a null pointer or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is executed.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execl	<process.h>	<errno.h>
_wexecl	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execle

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execle instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execle, _wexecle

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execle(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wexecle(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL,
   const char *const *envp
);
```

Parameters

cmdname
Path of the file to execute.

arg0, ... argn
List of pointers to parameters.

envp
Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space that's required for the arguments and the environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.

ERRNO VALUE	DESCRIPTION
EMFILE	Too many files are open. (The specified file must be opened to determine whether it is executable.)
ENOENT	The file or path is not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, which indicates that the calling process was not allocated correctly.

For more information about these return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, and passes each command-line argument as a separate parameter and passes an array of pointers to environment settings.

The **_execle** functions validate their parameters. If *cmdname* or *arg0* is a null pointer or an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execle	<process.h>	<errno.h>
_wexecle	<process.h> or <wchar.h>	<errno.h>

For more information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execlp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execlp instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execlp, _wexeclp

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execlp(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL
);
intptr_t _wexeclp(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

cmdname
Path of the file to execute.

arg0, ... *argn*
List of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.

ERRNO VALUE	DESCRIPTION
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter and using the **PATH** environment variable to find the file to execute.

The **_execlp** functions validate their parameters. If *cmdname* or *arg0* is a null pointer or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execlp	<process.h>	<errno.h>
_wexeclp	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execlpe

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execlpe instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execlpe, _wexeclpe

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execlpe(
   const char *cmdname,
   const char *arg0,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wexeclpe(
   const wchar_t *cmdname,
   const wchar_t *arg0,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

cmdname
Path of the file to execute.

arg0, ... argn
List of pointers to parameters.

envp
Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.

ERRNO VALUE	DESCRIPTION
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing each command-line argument as a separate parameter and also passing an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

The **_execlpe** functions validate their parameters. If either *cmdname* or *arg0* is a null pointers or empty string, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No new process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execlpe	<process.h>	<errno.h>
_wexeclpe	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execv

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execv instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execv, _wexecv

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execv(
   const char *cmdname,
   const char *const *argv
);
intptr_t _wexecv(
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

cmdname
Path of the file to execute.

argv
Array of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.

ERRNO VALUE	DESCRIPTION
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments.

The **_execv** functions validate their parameters. If *cmdname* is a null pointer, or if *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, the **_execv** functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execv	<process.h>	<errno.h>
_wexecv	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execve

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execve instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execve, _wexecve

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execve(
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wexecve(
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

cmdname
Path of the file to execute.

argv
Array of pointers to parameters.

envp
Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.

ERRNO VALUE	DESCRIPTION
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings.

_execve and **_wexecve** validate their parameters. If *cmdname* is a null pointer, or if *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execve	<process.h>	<errno.h>
_wexecve	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execvp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execvp instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execvp, _wexecvp

10/31/2018 • 2 minutes to read • Edit Online

Loads and executes new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execvp(
   const char *cmdname,
   const char *const *argv
);
intptr_t _wexecvp(
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

cmdname
Path of the file to execute.

argv
Array of pointers to parameters.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EINVAL	Invalid parameter.
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	The file or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.

ERRNO VALUE	DESCRIPTION
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, indicating that the calling process was not allocated properly.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, passing an array of pointers to command-line arguments and using the **PATH** environment variable to find the file to execute.

The **_execvp** functions validate their parameters. If the *cmdname* is a null pointer, or *argv* is a null pointer, pointer to an empty array, or if the array contains an empty string as the first argument, these functions invoke the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execvp	<process.h>	<errno.h>
_wexecvp	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

execvpe

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _execvpe instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_execvpe, _wexecvpe

10/31/2018 • 2 minutes to read • Edit Online

Loads and runs new child processes.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _execvpe(
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wexecvpe(
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

cmdname
Path of the file to execute.

argv
Array of pointers to parameters.

envp
Array of pointers to environment settings.

Return Value

If successful, these functions do not return to the calling process. A return value of -1 indicates an error, in which case the **errno** global variable is set.

ERRNO VALUE	DESCRIPTION
E2BIG	The space that's required for the arguments and environment settings exceeds 32 KB.
EACCES	The specified file has a locking or sharing violation.
EMFILE	Too many files are open. (The specified file must be opened to determine whether it is executable.)
ENOENT	The file or path is not found.

ERRNO VALUE	DESCRIPTION
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process; the available memory has been corrupted; or an invalid block exists, which indicates that the calling process was not allocated correctly.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions loads and executes a new process, and passes an array of pointers to command-line arguments and an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

The **_execvpe** functions validate their parameters. If the *cmdname* is a null pointer, or if *argv* is a null pointer, a pointer to an empty array, or a pointer to an array that contains an empty string as the first argument, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1. No process is launched.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_execvpe	<process.h>	<errno.h>
_wexecvpe	<process.h> or <wchar.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

See the example in _exec, _wexec Functions.

See also

Process and Environment Control
_exec, _wexec Functions
abort
atexit
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions
system, _wsystem

exit, _Exit, _exit

10/31/2018 • 2 minutes to read • Edit Online

Terminates the calling process. The **exit** function terminates it after cleanup; **_exit** and **_Exit** terminate it immediately.

> **NOTE**
> Do not use this method to shut down a Universal Windows Platform (UWP) app, except in testing or debugging scenarios. Programmatic or UI ways to close a Store app are not permitted according to the Microsoft Store policies. For more information, see UWP App lifecycle. For more information about Windows 10 apps, see How-to guides for Windows 10 apps.

Syntax

```
void exit(
   int const status
);
void _Exit(
   int const status
);
void _exit(
   int const status
);
```

Parameters

status
Exit status code.

Remarks

The **exit**, **_Exit** and **_exit** functions terminate the calling process. The **exit** function calls destructors for thread-local objects, then calls—in last-in-first-out (LIFO) order—the functions that are registered by **atexit** and **_onexit**, and then flushes all file buffers before it terminates the process. The **_Exit** and **_exit** functions terminate the process without destroying thread-local objects or processing **atexit** or **_onexit** functions, and without flushing stream buffers.

Although the **exit**, **_Exit** and **_exit** calls do not return a value, the value in *status* is made available to the host environment or waiting calling process, if one exists, after the process exits. Typically, the caller sets the *status* value to 0 to indicate a normal exit, or to some other value to indicate an error. The *status* value is available to the operating-system batch command **ERRORLEVEL** and is represented by one of two constants: **EXIT_SUCCESS**, which represents a value of 0, or **EXIT_FAILURE**, which represents a value of 1.

The **exit**, **_Exit**, **_exit**, **quick_exit**, **_cexit**, and **_c_exit** functions behave as follows.

FUNCTION	DESCRIPTION

FUNCTION	DESCRIPTION
exit	Performs complete C library termination procedures, terminates the process, and provides the supplied status code to the host environment.
_Exit	Performs minimal C library termination procedures, terminates the process, and provides the supplied status code to the host environment.
_exit	Performs minimal C library termination procedures, terminates the process, and provides the supplied status code to the host environment.
quick_exit	Performs quick C library termination procedures, terminates the process, and provides the supplied status code to the host environment.
_cexit	Performs complete C library termination procedures and returns to the caller. Does not terminate the process.
_c_exit	Performs minimal C library termination procedures and returns to the caller. Does not terminate the process.

When you call the **exit**, **_Exit** or **_exit** function, the destructors for any temporary or automatic objects that exist at the time of the call are not called. An automatic object is a non-static local object defined in a function. A temporary object is an object that's created by the compiler, such as a value returned by a function call. To destroy an automatic object before you call **exit**, **_Exit**, or **_exit**, explicitly call the destructor for the object, as shown here:

```
void last_fn() {}
    struct SomeClass {} myInstance{};
    // ...
    myInstance.~SomeClass(); // explicit destructor call
    exit(0);
}
```

Do not use **DLL_PROCESS_ATTACH** to call **exit** from **DllMain**. To exit the **DLLMain** function, return **FALSE** from **DLL_PROCESS_ATTACH**.

Requirements

FUNCTION	REQUIRED HEADER
exit, **_Exit**, **_exit**	<process.h> or <stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_exit.c
// This program returns an exit code of 1. The
// error code could be tested in a batch file.

#include <stdlib.h>

int main( void )
{
   exit( 1 );
}
```

See also

Process and Environment Control
abort
atexit
_cexit, _c_exit
_exec, _wexec Functions
_onexit, _onexit_m
quick_exit
_spawn, _wspawn Functions
system, _wsystem

exp, expf, expl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the exponential.

Syntax

```
double exp(
   double x
);
float exp(
   float x
);  // C++ only
long double exp(
   long double x
);  // C++ only
float expf(
   float x
);
long double expl(
   long double x
);
```

Parameters

x
The floating-point value to exponentiate the natural logarithm base *e* by.

Return Value

The **exp** functions return the exponential value of the floating-point parameter, *x*, if successful. That is, the result is e^x, where *e* is the base of the natural logarithm. On overflow, the function returns INF (infinity) and on underflow, **exp** returns 0.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± Quiet NaN, indeterminate	None	_DOMAIN
± Infinity	INVALID	_DOMAIN
x ≥ 7.097827e+002	INEXACT+OVERFLOW	OVERFLOW
X ≤ -7.083964e+002	INEXACT+UNDERFLOW	UNDERFLOW

The **exp** function has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **exp** that take a **float** or **long double** argument. In a C program, **exp** always takes and returns a **double**.

Requirements

FUNCTION	REQUIRED C HEADER	REQUIRED C++ HEADER
exp, **expf**, **expl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_exp.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.302585093, y;

   y = exp( x );
   printf( "exp( %f ) = %f\n", x, y );
}
```

```
exp( 2.302585 ) = 10.000000
```

See also

Floating-Point Support
log, logf, log10, log10f
_CIexp

exp2, exp2f, exp2l

10/31/2018 • 2 minutes to read • Edit Online

Computes 2 raised to the specified value.

Syntax

```
double exp2(
   double x
);

float exp2(
   float x
);  // C++ only

long double exp2(
   long double x
); // C++ only

float exp2f(
   float x
);

long double exp2l(
   long double x
);
```

Parameters

x
The value of the exponent.

Return Value

If successful, returns the base-2 exponent of *x*, that is, 2^x. Otherwise, it returns one of the following values:

ISSUE	RETURN
x = ±0	1
x = -INFINITY	+0
x = +INFINITY	+INFINITY
x = NaN	NaN
Overflow range error	+HUGE_VAL, +HUGE_VALF, or +HUGE_VALL
Underflow range error	Correct result, after rounding

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **exp2** that take and return **float** and **long double** types. In a C program, **exp2** always takes and returns a **double**.

Requirements

ROUTINE	C HEADER	C++ HEADER
exp, **expf**, **expl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
exp, expf, expl
log2, log2f, log2l

_expand

10/31/2018 • 2 minutes to read • Edit Online

Changes the size of a memory block.

Syntax

```
void *_expand(
   void *memblock,
   size_t size
);
```

Parameters

memblock
Pointer to previously allocated memory block.

size
New size in bytes.

Return Value

_expand returns a void pointer to the reallocated memory block. **_expand**, unlike **realloc**, cannot move a block to change its size. Thus, if there is sufficient memory available to expand the block without moving it, the *memblock* parameter to **_expand** is the same as the return value.

_expand returns **NULL** when an error is detected during its operation. For example, if **_expand** is used to shrink a memory block, it might detect corruption in the small block heap or an invalid block pointer and return **NULL**.

If there is insufficient memory available to expand the block to the given size without moving it, the function returns **NULL**. **_expand** never returns a block expanded to a size less than requested. If a failure occurs, **errno** indicates the nature of the failure. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To check the new size of the item, use **_msize**. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **_expand** function changes the size of a previously allocated memory block by trying to expand or contract the block without moving its location in the heap. The *memblock* parameter points to the beginning of the block. The *size* parameter gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes. *memblock* should not be a block that has been freed.

> **NOTE**
> On 64-bit platforms, **_expand** might not contract the block if the new size is less than the current size; in particular, if the block was less than 16K in size and therefore allocated in the Low Fragmentation Heap, **_expand** leaves the block unchanged and returns *memblock*.

When the application is linked with a debug version of the C run-time libraries, **_expand** resolves to _expand_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

This function validates its parameters. If *memblock* is a null pointer, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. If *size* is greater than **_HEAP_MAXREQ**, **errno** is set to **ENOMEM** and the function returns **NULL**.

Requirements

FUNCTION	REQUIRED HEADER
_expand	<malloc.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_expand.c

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main( void )
{
   char *bufchar;
   printf( "Allocate a 512 element buffer\n" );
   if( (bufchar = (char *)calloc( 512, sizeof( char ) )) == NULL )
      exit( 1 );
   printf( "Allocated %d bytes at %Fp\n",
         _msize( bufchar ), (void *)bufchar );
   if( (bufchar = (char *)_expand( bufchar, 1024 )) == NULL )
      printf( "Can't expand" );
   else
      printf( "Expanded block to %d bytes at %Fp\n",
            _msize( bufchar ), (void *)bufchar );
   // Free memory
   free( bufchar );
   exit( 0 );
}
```

```
Allocate a 512 element buffer
Allocated 512 bytes at 002C12BC
Expanded block to 1024 bytes at 002C12BC
```

See also

Memory Allocation
calloc
free
malloc
_msize
realloc

_expand_dbg

10/31/2018 • 3 minutes to read • Edit Online

Resizes a specified block of memory in the heap by expanding or contracting the block (debug version only).

Syntax

```
void *_expand_dbg(
   void *userData,
   size_t newSize,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData
Pointer to the previously allocated memory block.

newSize
Requested new size for the block (in bytes).

blockType
Requested type for resized block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested expand operation or **NULL**.

linenumber
Line number in the source file where the expand operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_expand_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, **_expand_dbg** returns a pointer to the resized memory block. Because the memory is not moved, the address is the same as the userData. If an error occurred or the block could not be expanded to the requested size, it returns **NULL**. If a failure occurs, **errno** is with information from the operating system about the nature of the failure. For more information about **errno**, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_expand_dbg** function is a debug version of the _expand function. When _DEBUG is not defined, each call to **_expand_dbg** is reduced to a call to **_expand**. Both **_expand** and **_expand_dbg** resize a memory block in the base heap, but **_expand_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename/linenumber* information to determine the origin of allocation requests.

_expand_dbg resizes the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header

information and overwrite buffers. The resize is accomplished by either expanding or contracting the original memory block. **_expand_dbg** does not move the memory block, as does the _realloc_dbg function.

When *newSize* is greater than the original block size, the memory block is expanded. During an expansion, if the memory block cannot be expanded to accommodate the requested size, **NULL** is returned. When *newSize* is less than the original block size, the memory block is contracted until the new size is obtained.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

This function validates its parameters. If *memblock* is a null pointer, or if size is greater than **_HEAP_MAXREQ**, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_expand_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_expand_dbg.c
//
// This program allocates a block of memory using _malloc_dbg
// and then calls _msize_dbg to display the size of that block.
// Next, it uses _expand_dbg to expand the amount of
// memory used by the buffer and then calls _msize_dbg again to
// display the new amount of memory allocated to the buffer.
//

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <crtdbg.h>

int main( void )
{
   long *buffer;
   size_t size;

   // Call _malloc_dbg to include the filename and line number
   // of our allocation request in the header
   buffer = (long *)_malloc_dbg( 40 * sizeof(long),
                                 _NORMAL_BLOCK, __FILE__, __LINE__ );
   if( buffer == NULL )
      exit( 1 );

   // Get the size of the buffer by calling _msize_dbg
   size = _msize_dbg( buffer, _NORMAL_BLOCK );
   printf( "Size of block after _malloc_dbg of 40 longs: %u\n", size );

   // Expand the buffer using _expand_dbg and show the new size
   buffer = (long *)_expand_dbg( buffer, size + sizeof(long),
                                 _NORMAL_BLOCK, __FILE__, __LINE__ );

   if( buffer == NULL )
      exit( 1 );
   size = _msize_dbg( buffer, _NORMAL_BLOCK );
   printf( "Size of block after _expand_dbg of 1 more long: %u\n",
           size );

   free( buffer );
   exit( 0 );
}
```

```
Size of block after _malloc_dbg of 40 longs: 160
Size of block after _expand_dbg of 1 more long: 164
```

Comment

The output of this program depends on your computer's ability to expand all the sections. If all sections are expanded, the output is reflected in the Output section.

See also

Debug Routines
_malloc_dbg

expm1, expm1f, expm1l

10/31/2018 • 2 minutes to read • Edit Online

Computes the base-e exponential of a value, minus one.

Syntax

```
double expm1(
   double x
);
float expm1(
   float x
);  // C++ only
long double expm1(
   long double x
);  // C++ only
float expm1f(
   float x
);
long double expm1l(
   long double x
);
```

Parameters

x
The floating-point exponential value.

Return Value

The **expm1** functions return a floating-point value that represents e^x - 1, if successful. On overflow, **expm1** returns **HUGE_VAL**, **expm1f** returns **HUGE_VALF**, **expm1l** returns **HUGE_VALL**, and **errno** is set to **ERANGE**. For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Because C++ allows overloading, you can call overloads of **expm1** that take and return **float** and **long double** values. In a C program, **expm1** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER
expm1, **expm1f**, **expm1l**	<math.h>

For additional compatibility information, see Compatibility.

See also

Floating-Point Support
exp2, exp2f, exp2l
pow, powf, powl

fabs, fabsf, fabsl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the absolute value of the floating-point argument.

Syntax

```
double fabs(
   double x
);
float fabs(
   float x
); // C++ only
long double fabs(
   long double x
); // C++ only
float fabsf(
   float x
);
long double fabsl(
   long double x
);
```

Parameters

x
Floating-point value.

Return Value

The **fabs** functions return the absolute value of the argument *x*. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	none	_DOMAIN

Remarks

C++ allows overloading, so you can call overloads of **fabs** if you include the <cmath> header. In a C program, **fabs** always takes and returns a **double**.

Requirements

FUNCTION	REQUIRED C HEADER	REQUIRED C++ HEADER
fabs, **fabsf**, **fabsl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

See the example for abs.

See also

Floating-Point Support
abs, labs, llabs, _abs64
_cabs

fclose, _fcloseall

10/31/2018 • 2 minutes to read • Edit Online

Closes a stream (**fclose**) or closes all open streams (**_fcloseall**).

Syntax

```
int fclose(
   FILE *stream
);
int _fcloseall( void );
```

Parameters

stream
Pointer to **FILE** structure.

Return Value

fclose returns 0 if the stream is successfully closed. **_fcloseall** returns the total number of streams closed. Both functions return **EOF** to indicate an error.

Remarks

The **fclose** function closes *stream*. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **fclose** sets **errno** to **EINVAL** and returns **EOF**. It is recommended that the *stream* pointer always be checked prior to calling this function.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

The **_fcloseall** function closes all open streams except **stdin**, **stdout**, **stderr** (and, in MS-DOS, **_stdaux** and **_stdprn**). It also closes and deletes any temporary files created by **tmpfile**. In both functions, all buffers associated with the stream are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned by the user with **setbuf** and **setvbuf** are not automatically released.

Note: When these functions are used to close a stream, the underlying file descriptor and OS file handle (or socket) are closed, as well as the stream. Thus, if the file was originally opened as a file handle or file descriptor and is closed with **fclose**, do not also call **_close** to close the file descriptor; do not call the Win32 function **CloseHandle** to close the file handle.

fclose and **_fcloseall** include code to protect against interference from other threads. For non-locking version of a **fclose**, see **_fclose_nolock**.

Requirements

FUNCTION	REQUIRED HEADER
fclose	<stdio.h>
_fcloseall	<stdio.h>

For additional compatibility information, see Compatibility.

Example

See the example for fopen.

See also

Stream I/O
_close
_fdopen, _wfdopen
fflush
fopen, _wfopen
freopen, _wfreopen

_fclose_nolock

10/31/2018 • 2 minutes to read • Edit Online

Closes a stream without thread-locking.

Syntax

```
int _fclose_nolock(
   FILE *stream
);
```

Parameters

stream
Pointer to the **FILE** structure.

Return Value

fclose returns 0 if the stream is successfully closed. Returns **EOF** to indicate an error.

Remarks

This functions is a non-locking version of **fclose**. It is identical except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fclose_nolock	<stdio.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
_close
_fdopen, _wfdopen
fflush
fopen, _wfopen
freopen, _wfreopen

fcloseall

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fcloseall instead.

fcvt

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fcvt or security-enhanced _fcvt_s instead.

_fcvt

10/31/2018 • 2 minutes to read • Edit Online

Converts a floating-point number to a string. A more secure version of this function is available; see _fcvt_s.

Syntax

```
char *_fcvt(
   double value,
   int count,
   int *dec,
   int *sign
);
```

Parameters

value
Number to be converted.

count
Number of digits after the decimal point.

dec
Pointer to the stored decimal-point position.

sign
Pointer to the stored sign indicator.

Return Value

_fcvt returns a pointer to the string of digits, **NULL** on error.

Remarks

The **_fcvt** function converts a floating-point number to a null-terminated character string. The *value* parameter is the floating-point number to be converted. **_fcvt** stores the digits of *value* as a string and appends a null character ('\0'). The *count* parameter specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, the string is padded with zeros.

The total number of digits returned by **_fcvt** will not exceed **_CVTBUFSIZE**.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and sign after the call. The *dec* parameter points to an integer value; this integer value gives the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The parameter *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

The difference between **_ecvt** and **_fcvt** is in the interpretation of the *count* parameter. **_ecvt** interprets *count* as the total number of digits in the output string, whereas **_fcvt** interprets *count* as the number of digits after the decimal point.

_ecvt and **_fcvt** use a single statically allocated buffer for the conversion. Each call to one of these routines

destroys the results of the previous call.

This function validates its parameters. If *dec* or *sign* is **NULL**, or *count* is 0, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and **NULL** is returned.

Requirements

FUNCTION	REQUIRED HEADER
_fcvt	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fcvt.c
// compile with: /W3
// This program converts the constant
// 3.1415926535 to a string and sets the pointer
// buffer to point to that string.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int  decimal, sign;
   char *buffer;
   double source = 3.1415926535;

   buffer = _fcvt( source, 7, &decimal, &sign ); // C4996
   // Note: _fcvt is deprecated; consider using _fcvt_s instead
   printf( "source: %2.10f   buffer: '%s'   decimal: %d   sign: %d\n",
            source, buffer, decimal, sign );
}
```

```
source: 3.1415926535   buffer: '31415927'   decimal: 1   sign: 0
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt
_gcvt

_fcvt_s

10/31/2018 • 3 minutes to read • Edit Online

Converts a floating-point number to a string. This is a version of _fcvt with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _fcvt_s(
   char* buffer,
   size_t sizeInBytes,
   double value,
   int count,
   int *dec,
   int *sign
);
template <size_t size>
errno_t _fcvt_s(
   char (&buffer)[size],
   double value,
   int count,
   int *dec,
   int *sign
); // C++ only
```

Parameters

buffer
The supplied buffer that will hold the result of the conversion.

sizeInBytes
The size of the buffer in bytes.

value
Number to be converted.

count
Number of digits after the decimal point.

dec
Pointer to the stored decimal-point position.

sign
Pointer to the stored sign indicator.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For a listing of these errors, see errno, _doserrno, _sys_errlist, and _sys_nerr.

In the case of an invalid parameter, as listed in the following table, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

BUFFER	*SIZEINBYTES*	VALUE	COUNT	DEC	SIGN	RETURN	VALUE IN *BUFFER*
NULL	any	any	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	<=0	any	any	any	any	**EINVAL**	Not modified.
any	any	any	any	**NULL**	any	**EINVAL**	Not modified.
any	any	any	any	any	**NULL**	**EINVAL**	Not modified.

Security Issues

_fcvt_s might generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_fcvt_s** function converts a floating-point number to a null-terminated character string. The *value* parameter is the floating-point number to be converted. **_fcvt_s** stores the digits of *value* as a string and appends a null character ('\0'). The *count* parameter specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value; this integer value gives the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The parameter *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

A buffer of length **_CVTBUFSIZE** is sufficient for any floating point value.

The difference between **_ecvt_s** and **_fcvt_s** is in the interpretation of the *count* parameter. **_ecvt_s** interprets *count* as the total number of digits in the output string, and **_fcvt_s** interprets *count* as the number of digits after the decimal point.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_fcvt_s	<stdlib.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries: All versions of the CRT Library Features.

Example

```
// fcvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    char * buf = 0;
    int decimal;
    int sign;
    int err;

    buf = (char*) malloc(_CVTBUFSIZE);
    err = _fcvt_s(buf, _CVTBUFSIZE, 1.2, 5, &decimal, &sign);

    if (err != 0)
    {
        printf("_fcvt_s failed with error code %d\n", err);
        exit(1);
    }

    printf("Converted value: %s\n", buf);
}
```

```
Converted value: 120000
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt_s
_gcvt_s
_fcvt

fdim, fdimf, fdiml

11/9/2018 • 2 minutes to read • Edit Online

Determines the positive difference between the first and second values.

Syntax

```
double fdim(
   double x,
   double y
);

float fdim(
   float x,
   float y
); //C++ only

long double fdim(
   long double x,
   long double y
); //C++ only

float fdimf(
   float x,
   float y
);

long double fdiml(
   long double x,
   long double y
);
```

Parameters

x
The first value.

y
The second value.

Return Value

Returns the positive difference between *x* and *y*:

RETURN VALUE	SCENARIO
x-y	if x > y
0	if x <= y

Otherwise, may return one of the following errors:

ISSUE	RETURN
Overflow range error	+HUGE_VAL, +HUGE_VALF, or +HUGE_VALL

ISSUE	RETURN
Underflow range error	correct value (after rounding)
x or *y* is NaN	NaN

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **fdim** that take and return **float** and **long double** types. In a C program, **fdim** always takes and returns a **double**.

Except for the NaN handling, this function is equivalent to `fmax(x - y, 0)`.

Requirements

FUNCTION	C HEADER	C++ HEADER
fdim, **fdimf**, **fdiml**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fmax, fmaxf, fmaxl
abs, labs, llabs, _abs64

fdopen

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fdopen instead.

_fdopen, _wfdopen

11/8/2018 • 4 minutes to read • Edit Online

Associates a stream with a file that was previously opened for low-level I/O.

Syntax

```
FILE *_fdopen(
   int fd,
   const char *mode
);
FILE *_wfdopen(
   int fd,
   const wchar_t *mode
);
```

Parameters

fd
File descriptor of the open file.

mode
Type of file access.

Return Value

Each of these functions returns a pointer to the open stream. A null pointer value indicates an error. When an error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set either to **EBADF**, which indicates a bad file descriptor, or **EINVAL**, which indicates that *mode* was a null pointer.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_fdopen** function associates an I/O stream with the file that is identified by *fd*, and thus allows a file that is opened for low-level I/O to be buffered and formatted. **_wfdopen** is a wide-character version of **_fdopen**; the *mode* argument to **_wfdopen** is a wide-character string. **_wfdopen** and **_fdopen** otherwise behave identically.

File descriptors passed into **_fdopen** are owned by the returned **FILE *** stream. If **_fdopen** is successful, do not call _close on the file descriptor. Calling fclose on the returned **FILE *** also closes the file descriptor.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfdopen	**_fdopen**	**_fdopen**	**_wfdopen**

The *mode* character string specifies the type of file access requested for the file:

MODE	ACCESS
"r"	Opens for reading. If the file does not exist or cannot be found, the **fopen** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending). Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. Creates the file if it does not exist.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned by using fseek or rewind, but it is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten. When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fflush, fsetpos, fseek, or rewind operation. You can specify the current position for the fsetpos or fseek operation, if you want to.

In addition to the above values, the following characters can also be included in *mode* to specify the translation mode for newline characters:

MODE MODIFIER	BEHAVIOR
t	Open in text (translated) mode. In this mode, carriage return-line feed (CR-LF) combinations are translated into one-line feeds (LF) on input, and LF characters are translated to CR-LF combinations on output. Also, Ctrl+Z is interpreted as an end-of-file character on input.
b	Open in binary (untranslated) mode. Any translations from **t** mode are suppressed.
c	Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.
n	Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with Commode.obj. The global commit flag default is "no-commit" unless you explicitly link your program with Commode.obj.

The **t**, **c**, and **n** *mode* options are Microsoft extensions for **fopen** and **_fdopen**. Do not use them if you want to preserve ANSI portability.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns NULL. For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Valid characters for the *mode* string used in **fopen** and **_fdopen** correspond to *oflag* arguments used in _open and _sopen, as shown in this table:

CHARACTERS IN *MODE* STRING	EQUIVALENT *OFLAG* VALUE FOR _OPEN AND _SOPEN
a	**_O_WRONLY | _O_APPEND** (usually **_O_WRONLY | _O_CREAT | _O_APPEND**)
a+	**_O_RDWR | _O_APPEND** (usually **_O_RDWR | _O_APPEND | _O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**
w	**_O_WRONLY** (usually **_O_WRONLY | _O_CREAT | _O_TRUNC**)
w+	**_O_RDWR** (usually **_O_RDWR | _O_CREAT | _O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None

Requirements

FUNCTION	REQUIRED HEADER
_fdopen	<stdio.h>
_wfdopen	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fdopen.c
// This program opens a file by using low-level
// I/O, then uses _fdopen to switch to stream
// access. It counts the lines in the file.

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <share.h>

int main( void )
{
   FILE *stream;
   int  fd, count = 0;
   char inbuf[128];

   // Open a file.
   if( _sopen_s( &fd, "crt_fdopen.txt", _O_RDONLY, _SH_DENYNO, 0 ) )
      exit( 1 );

   // Get stream from file descriptor.
   if( (stream = _fdopen( fd, "r" )) == NULL )
      exit( 1 );

   while( fgets( inbuf, 128, stream ) != NULL )
      count++;

   // After _fdopen, close by using fclose, not _close.
   fclose( stream );
   printf( "Lines in file: %d\n", count );
}
```

Input: crt_fdopen.txt

```
Line one
Line two
```

Output

```
Lines in file: 2
```

See also

Stream I/O
_dup, _dup2
fclose, _fcloseall
fopen, _wfopen
freopen, _wfreopen
_open, _wopen

feclearexcept

10/31/2018 • 2 minutes to read • Edit Online

Attempts to clear the floating-point exception flags specified by the argument.

Syntax

```
int feclearexcept(
   int excepts
);
```

Parameters

excepts
The exception status flags to clear.

Return Value

Returns zero if *excepts* is zero, or if all the specified exceptions were successfully cleared. Otherwise, returns a nonzero value.

Remarks

The **feclearexcept** function attempts to clear the floating point exception status flags specified by *excepts*. The function supports these exception macros, defined in fenv.h:

EXCEPTION MACRO	DESCRIPTION
FE_DIVBYZERO	A singularity or pole error occurred in an earlier floating-point operation; an infinity value was created.
FE_INEXACT	The function was forced to round the stored result of an earlier floating-point operation.
FE_INVALID	A domain error occurred in an earlier floating-point operation.
FE_OVERFLOW	A range error occurred; an earlier floating-point operation result was too large to be represented.
FE_UNDERFLOW	An earlier floating-point operation result was too small to be represented at full precision; a denormal value was created.
FE_ALL_EXCEPT	The bitwise OR of all supported floating-point exceptions.

The *excepts* argument may be zero, or the bitwise OR of one or more of the supported exception macros. The result of any other argument value is undefined.

Requirements

FUNCTION	C HEADER	C++ HEADER
feclearexcept	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fetestexcept

fegetenv

10/31/2018 • 2 minutes to read • Edit Online

Stores the current floating-point environment in the specified object.

Syntax

```
int fegetenv(
   fenv_t *penv
);
```

Parameters

penv
Pointer to an **fenv_t** object to contain the current floating-point environment values.

Return Value

Returns 0 if the floating-point environment was successfully stored in *penv*. Otherwise, returns a non-zero value.

Remarks

The **fegetenv** function stores the current floating-point environment in the object pointed to by *penv*. The floating point environment is the set of status flags and control modes that affect floating-point calculations. This includes the rounding direction mode and the status flags for floating-point exceptions. If *penv* does not point to a valid **fenv_t** object, subsequent behavior is undefined.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fegetenv	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fesetenv

fegetexceptflag

11/9/2018 • 2 minutes to read • Edit Online

Stores the current state of the specified floating-point exception flags.

Syntax

```
int fegetexceptflag(
   fexcept_t* pstatus,
   int excepts
);
```

Parameters

pstatus
A pointer to a **fexcept_t** object to contain the current values of the exception flags specified by *excepts*.

excepts
The floating-point exception flags to store in *pstatus*.

Return Value

On success, returns 0. Otherwise, returns a non-zero value.

Remarks

The **fegetexceptflag** function stores the current state of the floating-point exception status flags specified by *excepts* in the **fexcept_t** object pointed to by *pstatus*. *pstatus* must point to a valid **fexcept_t** object, or subsequent behavior is undefined. The **fegetexceptflag** function supports these exception macros, defined in <fenv.h>:

EXCEPTION MACRO	DESCRIPTION
FE_DIVBYZERO	A singularity or pole error occurred in an earlier floating-point operation; an infinity value was created.
FE_INEXACT	The function was forced to round the stored result of an earlier floating-point operation.
FE_INVALID	A domain error occurred in an earlier floating-point operation.
FE_OVERFLOW	A range error occurred; an earlier floating-point operation result was too large to be represented.
FE_UNDERFLOW	An earlier floating-point operation result was too small to be represented at full precision; a denormal value was created.
FE_ALLEXCEPT	The bitwise OR of all supported floating-point exceptions.

The *excepts* argument may be zero, one of the supported floating-point exception macros, or the bitwise OR of two or more of the macros. The effect of any other argument value is undefined.

To use this function, you must turn off floating-point optimizations that could prevent access by using the

`#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fegetexceptflag	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fesetexceptflag

fegetround, fesetround

10/31/2018 • 2 minutes to read • Edit Online

Gets or sets the current floating-point rounding mode.

Syntax

```
int fegetround(void);

int fesetround(
   int round_mode
);
```

Parameters

round_mode
The rounding mode to set, as one of the floating-point rounding macros. If the value is not equal to one of the floating-point rounding macros, the rounding mode is not changed.

Return Value

On success, **fegetround** returns the rounding mode as one of the floating point rounding macro values. It returns a negative value if the current rounding mode can't be determined.

On success, **fesetround** returns 0. Otherwise, a non-zero value is returned.

Remarks

Floating-point operations can use one of several rounding modes. These control which direction the results of floating-point operations are rounded toward when the results are stored. These are the names and behaviors of the floating-point rounding macros defined in <fenv.h>:

MACRO	DESCRIPTION
FE_DOWNWARD	Round towards negative infinity.
FE_TONEAREST	Round towards the nearest.
FE_TOWARDZERO	Round towards zero.
FE_UPWARD	Round towards positive infinity.

The default behavior of FE_TONEAREST is to round results midway between representable values toward the nearest value with an even (0) least significant bit.

The current rounding mode affects these operations:

- String conversions.
- The results of floating-point arithmetic operators outside of constant expressions.
- The library rounding functions, such as **rint** and **nearbyint**.

- Return values from standard library mathematical functions.

The current rounding mode does not affect these operations:

- The **trunc**, **ceil**, **floor**, and **lround** library functions.
- Floating-point to integer implicit casts and conversions, which always round towards zero.
- The results of floating-point arithmetic operators in constant expressions, which always round to the nearest value.

To use these functions, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fegetround, **fesetround**	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
nearbyint, nearbyintf, nearbyintl
rint, rintf, rintl
lrint, lrintf, lrintl, llrint, llrintf, llrintl

feholdexcept

10/31/2018 • 2 minutes to read • Edit Online

Saves the current floating-point environment in the specified object, clears the floating-point status flags, and, if possible, puts the floating-point environment into non-stop mode.

Syntax

```
int feholdexcept(
   fenv_t *penv
);
```

Parameters

penv
Pointer to an **fenv_t** object to contain a copy of the floating-point environment.

Return Value

Returns zero if and only if the function is able to successfully turn on non-stop floating-point exception handling.

Remarks

The **feholdexcept** function is used to store the state of the current floating point environment in the **fenv_t** object pointed to by *penv*, and to set the environment to not interrupt execution on floating-point exceptions. This is known as non-stop mode. This mode continues until the environment is restored using fesetenv or feupdateenv.

You can use this function at the beginning of a subroutine that needs to hide one or more floating-point exceptions from the caller. To report an exception, you can simply clear the unwanted exceptions by using feclearexcept, and then end the non-stop mode with a call to **feupdateenv**.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
feholdexcept	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
feclearexcept
fesetenv
feupdateenv

feof

10/31/2018 • 2 minutes to read • Edit Online

Tests for end-of-file on a stream.

Syntax

```
int feof(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Return Value

The **feof** function returns a nonzero value if a read operation has attempted to read past the end of the file; it returns 0 otherwise. If the stream pointer is **NULL**, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the **feof** returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **feof** routine (implemented both as a function and as a macro) determines whether the end of *stream* has been passed. When the end of file is passed, read operations return an end-of-file indicator until the stream is closed or until rewind, **fsetpos**, fseek, or **clearerr** is called against it.

For example, if a file contains 10 bytes and you read 10 bytes from the file, **feof** will return 0 because, even though the file pointer is at the end of the file, you have not attempted to read beyond the end. Only after you try to read an 11th byte will **feof** return a nonzero value.

Requirements

FUNCTION	REQUIRED HEADER
feof	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_feof.c
// This program uses feof to indicate when
// it reaches the end of the file CRT_FEOF.TXT. It also
// checks for errors with ferror.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int  count, total = 0;
   char buffer[100];
   FILE *stream;

   fopen_s( &stream, "crt_feof.txt", "r" );
   if( stream == NULL )
      exit( 1 );

   // Cycle until end of file reached:
   while( !feof( stream ) )
   {
      // Attempt to read in 100 bytes:
      count = fread( buffer, sizeof( char ), 100, stream );
      if( ferror( stream ) )      {
         perror( "Read error" );
         break;
      }

      // Total up actual bytes read
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( stream );
}
```

Input: crt_feof.txt

```
Line one.
Line two.
```

Output

```
Number of bytes read = 19
```

See also

Error Handling
Stream I/O
clearerr
_eof
ferror
perror, _wperror

feraiseexcept

10/31/2018 • 2 minutes to read • Edit Online

Raises the specified floating-point exceptions.

Syntax

```
int feraiseexcept(
   int excepts
);
```

Parameters

excepts
The floating-point exceptions to raise.

Return Value

If all specified exceptions are raised successfully, returns 0.

Remarks

The **feraiseexcept** function attempts to raise the floating-point exceptions specified by *excepts*. The **feraiseexcept** function supports these exception macros, defined in <fenv.h>:

EXCEPTION MACRO	DESCRIPTION
FE_DIVBYZERO	A singularity or pole error occurred in an earlier floating-point operation; an infinity value was created.
FE_INEXACT	The function was forced to round the stored result of an earlier floating-point operation.
FE_INVALID	A domain error occurred in an earlier floating-point operation.
FE_OVERFLOW	A range error occurred; an earlier floating-point operation result was too large to be represented.
FE_UNDERFLOW	An earlier floating-point operation result was too small to be represented at full precision; a denormal value was created.
FE_ALLEXCEPT	The bitwise OR of all supported floating-point exceptions.

The *excepts* argument may be zero, one of the exception macro values, or the bitwise OR of two or more of the supported exception macros. If one of the specified exception macros is FE_OVERFLOW or FE_UNDERFLOW, the FE_INEXACT exception may be raised as a side-effect.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Microsoft Specific: The exceptions specified in *excepts* are raised in the order FE_INVALID, FE_DIVBYZERO,

FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT. However, FE_INEXACT can be raised when FE_OVERFLOW or FE_UNDERFLOW is raised, even if not specified in *excepts*. **End Microsoft Specific**

Requirements

FUNCTION	C HEADER	C++ HEADER
feraiseexcept	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fesetexceptflag
feholdexcept
fetestexcept
feupdateenv

ferror

10/31/2018 • 2 minutes to read • Edit Online

Tests for an error on a stream.

Syntax

```
int ferror(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Return Value

If no error has occurred on *stream*, **ferror** returns 0. Otherwise, it returns a nonzero value. If stream is **NULL**, **ferror** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **ferror** routine (implemented both as a function and as a macro) tests for a reading or writing error on the file associated with *stream*. If an error has occurred, the error indicator for the stream remains set until the stream is closed or rewound, or until **clearerr** is called against it.

Requirements

FUNCTION	REQUIRED HEADER
ferror	<stdio.h>

For additional compatibility information, see Compatibility.

Example

See the example for feof.

See also

Error Handling
Stream I/O
clearerr
_eof
feof
fopen, _wfopen

perror, _wperror

fesetenv

10/31/2018 • 2 minutes to read • Edit Online

Sets the current floating-point environment.

Syntax

```
int fesetenv(
   const fenv_t *penv
);
```

Parameters

penv
Pointer to a **fenv_t** object that contains a floating-point environment as set by a call to fegetenv or feholdexcept. You can also specify the default startup floating-point environment by using the **FE_DFL_ENV** macro.

Return Value

Returns 0 if the environment was successfully set. Otherwise, returns a nonzero value.

Remarks

The **fesetenv** function sets the current floating-point environment from the value stored in the **fenv_t** object pointed to by *penv*. The floating point environment is the set of status flags and control modes that affect floating-point calculations. This includes the rounding mode and the status flags for floating-point exceptions. If *penv* is not **FE_DFL_ENV** or does not point to a valid **fenv_t** object, subsequent behavior is undefined.

A call to this function sets the exception status flags that are in the *penv* object, but it does not raise those exceptions.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fesetenv	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fegetenv
feclearexcept
feholdexcept
fesetexceptflag

fesetexceptflag

10/31/2018 • 2 minutes to read • Edit Online

Sets the specified floating-point status flags in the current floating-point environment.

Syntax

```
int fesetexceptflag(
     const fexcept_t *pstatus,
     int excepts
);
```

Parameters

pstatus
Pointer to an **fexcept_t** object containing the values to set the exception status flags to. The object may be set by a previous call to fegetexceptflag.

excepts
The floating-point exception status flags to set.

Return Value

If all the specified exception status flags are set successfully, returns 0. Otherwise, returns a nonzero value.

Remarks

The **fesetexceptflag** function sets the state of the floating-point exception status flags specified by *excepts* to the corresponding values set in the **fexcept_t** object pointed to by *pstatus*. It does not raise the exceptions. The *pstatus* pointer must point to a valid **fexcept_t** object, or subsequent behavior is undefined. The **fesetexceptflag** function supports these exception macro values in *excepts*, defined in <fenv.h>:

EXCEPTION MACRO	DESCRIPTION
FE_DIVBYZERO	A singularity or pole error occurred in an earlier floating-point operation; an infinity value was created.
FE_INEXACT	The function was forced to round the stored result of an earlier floating-point operation.
FE_INVALID	A domain error occurred in an earlier floating-point operation.
FE_OVERFLOW	A range error occurred; an earlier floating-point operation result was too large to be represented.
FE_UNDERFLOW	An earlier floating-point operation result was too small to be represented at full precision; a denormal value was created.
FE_ALLEXCEPT	The bitwise OR of all supported floating-point exceptions.

The *excepts* argument may be zero, one of the supported floating-point exception macros, or the bitwise OR of two or more of the macros. The effect of any other argument value is undefined.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fesetexceptflag	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fegetexceptflag

fetestexcept

11/9/2018 • 2 minutes to read • Edit Online

Determines which of the specified floating-point exception status flags are currently set.

Syntax

```
int fetestexcept(
   int excepts
);
```

Parameters

excepts
A bitwise OR of the floating-point status flags to test.

Return Value

On success, returns a bitmask containing a bitwise OR of the floating-point exception macros that correspond to the exception status flags currently set. Returns 0 if none of the exceptions are set.

Remarks

Use the fetestexcept function to determine which exceptions were raised by a floating point operation. Use the *excepts* parameter to specify which exception status flags to test. The **fetestexcept** function uses these exception macros defined in <fenv.h> in *excepts* and the return value:

EXCEPTION MACRO	DESCRIPTION
FE_DIVBYZERO	A singularity or pole error occurred in an earlier floating-point operation; an infinity value was created.
FE_INEXACT	The function was forced to round the stored result of an earlier floating-point operation.
FE_INVALID	A domain error occurred in an earlier floating-point operation.
FE_OVERFLOW	A range error occurred; an earlier floating-point operation result was too large to be represented.
FE_UNDERFLOW	An earlier floating-point operation result was too small to be represented at full precision; a denormal value was created.
FE_ALLEXCEPT	The bitwise OR of all supported floating-point exceptions.

The specified *excepts* argument may be 0, one of the supported floating-point exception macros, or the bitwise OR of two or more of the macros. The effect of any other *excepts* argument value is undefined.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
fetestexcept	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
feclearexcept
feraiseexcept

feupdateenv

10/31/2018 • 2 minutes to read • Edit Online

Saves the currently raised floating-point exceptions, restores the specified floating-point environment state, and then raises the saved floating-point exceptions.

Syntax

```
int feupdateenv(
   const fenv_t* penv
);
```

Parameters

penv
Pointer to a **fenv_t** object that contains a floating-point environment as set by a call to fegetenv or feholdexcept. You can also specify the default startup floating-point environment by using the FE_DFL_ENV macro.

Return Value

Returns 0 if all actions completed successfully. Otherwise, returns a nonzero value.

Remarks

The **feupdateenv** function performs multiple actions. First, it stores the current raised floating-point exception status flags in automatic storage. Then, it sets the current floating-point environment from the value stored in the **fenv_t** object pointed to by *penv*. If *penv* is not **FE_DFL_ENV** or does not point to a valid **fenv_t** object, subsequent behavior is undefined. Finally, **feupdateenv** raises the locally stored floating-point exceptions.

To use this function, you must turn off floating-point optimizations that could prevent access by using the `#pragma fenv_access(on)` directive prior to the call. For more information, see fenv_access.

Requirements

FUNCTION	C HEADER	C++ HEADER
feupdateenv	<fenv.h>	<cfenv>

For additional compatibility information, see Compatibility.

See also

fegetenv
feclearexcept
feholdexcept
fesetexceptflag

fflush

11/8/2018 • 2 minutes to read • Edit Online

Flushes a stream.

Syntax

```
int fflush(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Return Value

fflush returns 0 if the buffer was successfully flushed. The value 0 is also returned in cases in which the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.

> **NOTE**
> If **fflush** returns **EOF**, data may have been lost due to a write failure. When setting up a critical error handler, it is safest to turn buffering off with the **setvbuf** function or to use low-level I/O routines such as **_open**, **_close**, and **_write** instead of the stream I/O functions.

Remarks

The **fflush** function flushes the stream *stream*. If the stream was opened in write mode, or it was opened in update mode and the last operation was a write, the contents of the stream buffer are written to the underlying file or device and the buffer is discarded. If the stream was opened in read mode, or if the stream has no buffer, the call to **fflush** has no effect, and any buffer is retained. A call to **fflush** negates the effect of any prior call to **ungetc** for the stream. The stream remains open after the call.

If *stream* is **NULL**, the behavior is the same as a call to **fflush** on each open stream. All streams opened in write mode and all streams opened in update mode where the last operation was a write are flushed. The call has no effect on other streams.

Buffers are normally maintained by the operating system, which determines the optimal time to write the data automatically to disk: when a buffer is full, when a stream is closed, or when a program terminates normally without closing the stream. The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating-system buffers. Without rewriting an existing program, you can enable this feature by linking the program's object files with COMMODE.OBJ. In the resulting executable file, calls to **_flushall** write the contents of all buffers to disk. Only **_flushall** and **fflush** are affected by COMMODE.OBJ.

For information about controlling the commit-to-disk feature, see Stream I/O, fopen, and _fdopen.

This function locks the calling thread and is therefore thread-safe. For a non-locking version, see **_fflush_nolock**.

Requirements

FUNCTION	REQUIRED HEADER
fflush	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fflush.c
#include <stdio.h>
#include <conio.h>

int main( void )
{
   int integer;
   char string[81];

   // Read each word as a string.
   printf( "Enter a sentence of four words with scanf: " );
   for( integer = 0; integer < 4; integer++ )
   {
      scanf_s( "%s", string, sizeof(string) );
      printf( "%s\n", string );
   }

   // You must flush the input buffer before using gets.
   // fflush on input stream is an extension to the C standard
   fflush( stdin );
   printf( "Enter the same sentence with gets: " );
   gets_s( string, sizeof(string) );
   printf( "%s\n", string );
}
```

```
This is a test
This is a test
```

```
Enter a sentence of four words with scanf: This is a test
This
is
a
test
Enter the same sentence with gets: This is a test
This is a test
```

See also

Stream I/O
fclose, _fcloseall
_flushall
setvbuf

_fflush_nolock

10/31/2018 • 2 minutes to read • Edit Online

Flushes a stream without locking the thread.

Syntax

```
int _fflush_nolock(
   FILE *stream
);
```

Parameters

stream
Pointer to the **FILE** structure.

Return Value

See fflush.

Remarks

This function is a non-locking version of **fflush**. It is identical to **fflush** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fflush_nolock	<stdio.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
fclose, _fcloseall
_flushall
setvbuf

fgetc, fgetwc

10/31/2018 • 2 minutes to read • Edit Online

Read a character from a stream.

Syntax

```
int fgetc(
   FILE *stream
);
wint_t fgetwc(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Return Value

fgetc returns the character read as an **int** or returns **EOF** to indicate an error or end of file. **fgetwc** returns, as a wint_t, the wide character that corresponds to the character read or returns **WEOF** to indicate an error or end of file. For both functions, use **feof** or **ferror** to distinguish between an error and an end-of-file condition. If a read error occurs, the error indicator for the stream is set. If *stream* is **NULL**, **fgetc** and **fgetwc** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EOF**.

Remarks

Each of these functions reads a single character from the current position of the file associated with *stream*. The function then increments the associated file pointer (if defined) to point to the next character. If the stream is at end of file, the end-of-file indicator for the stream is set.

fgetc is equivalent to **getc**, but is implemented only as a function, rather than as a function and a macro.

fgetwc is the wide-character version of **fgetc**; it reads **c** as a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads.

For more information about processing wide characters and multibyte characters in text and binary modes, see Unicode Stream I/O in Text and Binary Modes.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fgettc	**fgetc**	**fgetc**	**fgetwc**

Requirements

FUNCTION	REQUIRED HEADER
fgetc	<stdio.h>
fgetwc	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fgetc.c
// This program uses getc to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   char buffer[81];
   int  i, ch;

   // Open file to read line from:
   fopen_s( &stream, "crt_fgetc.txt", "r" );
   if( stream == NULL )
      exit( 0 );

   // Read in first 80 characters and place them in "buffer":
   ch = fgetc( stream );
   for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = fgetc( stream );
   }

   // Add null to end string
   buffer[i] = '\0';
   printf( "%s\n", buffer );
   fclose( stream );
}
```

Input: crt_fgetc.txt

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

See also

_fgetc_nolock, _fgetwc_nolock

10/31/2018 • 2 minutes to read • Edit Online

Reads a character from a stream without locking the thread.

Syntax

```
int _fgetc_nolock(
   FILE *stream
);
wint_t _fgetwc_nolock(
   FILE *stream
);
```

Parameters

stream
Pointer to the **FILE** structure.

Return Value

Seefgetc, fgetwc.

Remarks

_fgetc_nolock and **_fgetwc_nolock** are identical to **fgetc** and **fgetwc**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fgettc_nolock	**_fgetc_nolock**	**_fgetc_nolock**	**_fgetwc_nolock**

Requirements

FUNCTION	REQUIRED HEADER
_fgetc_nolock	<stdio.h>
_fgetwc_nolock	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fgetc_nolock.c
// This program uses getc to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   char buffer[81];
   int  i, ch;

   // Open file to read line from:
   if( fopen_s( &stream, "crt_fgetc_nolock.txt", "r" ) != 0 )
      exit( 0 );

   // Read in first 80 characters and place them in "buffer":
   ch = fgetc( stream );
   for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = _fgetc_nolock( stream );
   }

   // Add null to end string
   buffer[i] = '\0';
   printf( "%s\n", buffer );
   fclose( stream );
}
```

Input: crt_fgetc_nolock.txt

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

See also

Stream I/O
fputc, fputwc
getc, getwc

fgetchar

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fgetchar instead.

_fgetchar, _fgetwchar

10/31/2018 • 2 minutes to read • Edit Online

Reads a character from **stdin**.

Syntax

```
int _fgetchar( void );
wint_t _fgetwchar( void );
```

Return Value

_fgetchar returns the character read as an **int** or returns `EOF` to indicate an error or end of file. **_fgetwchar** returns, as a wint_t, the wide character that corresponds to the character read or returns `WEOF` to indicate an error or end of file. For both functions, use **feof** or **ferror** to distinguish between an error and an end-of-file condition.

Remarks

These functions read a single character from **stdin**. The function then increments the associated file pointer (if defined) to point to the next character. If the stream is at end of file, the end-of-file indicator for the stream is set.

_fgetchar is equivalent to `fgetc( stdin )`. It is also equivalent to **getchar**, but implemented only as a function, rather than as a function and a macro. **_fgetwchar** is the wide-character version of **_fgetchar**.

These functions are not compatible with the ANSI standard.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fgettchar	**_fgetchar**	**_fgetchar**	**_fgetwchar**

Requirements

FUNCTION	REQUIRED HEADER
_fgetchar	<stdio.h>
_fgetwchar	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console—**stdin**, **stdout**, and **stderr**—must be redirected before C run-time functions can use them in UWP apps. For more compatibility information, see Compatibility.

Example

```
// crt_fgetchar.c
// This program uses _fgetchar to read the first
// 80 input characters (or until the end of input)
// and place them into a string named buffer.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char buffer[81];
   int  i, ch;

   // Read in first 80 characters and place them in "buffer":
   ch = _fgetchar();
   for( i=0; (i < 80 ) && ( feof( stdin ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = _fgetchar();
   }

   // Add null to end string
   buffer[i] = '\0';
   printf( "%s\n", buffer );
}
```

```
      Line one.
Line two.Line one.
Line two.
```

See also

Stream I/O

fputc, fputwc

getc, getwc

fgetpos

10/31/2018 • 2 minutes to read • Edit Online

Gets a stream's file-position indicator.

Syntax

```
int fgetpos(
   FILE *stream,
   fpos_t *pos
);
```

Parameters

stream
Target stream.

pos
Position-indicator storage.

Return Value

If successful, **fgetpos** returns 0. On failure, it returns a nonzero value and sets **errno** to one of the following manifest constants (defined in STDIO.H): **EBADF**, which means the specified stream is not a valid file pointer or is not accessible, or **EINVAL**, which means the *stream* value or the value of *pos* is invalid, such as if either is a null pointer. If *stream* or *pos* is a **NULL** pointer, the function invokes the invalid parameter handler, as described in Parameter Validation.

Remarks

The **fgetpos** function gets the current value of the *stream* argument's file-position indicator and stores it in the object pointed to by *pos*. The **fsetpos** function can later use information stored in *pos* to reset the *stream* argument's pointer to its position at the time **fgetpos** was called. The *pos* value is stored in an internal format and is intended for use only by **fgetpos** and **fsetpos**.

Requirements

FUNCTION	REQUIRED HEADER
fgetpos	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fgetpos.c
// This program uses fgetpos and fsetpos to
// return to a location in a file.

#include <stdio.h>

int main( void )
{
   FILE   *stream;
   fpos_t pos;
   char   buffer[20];

   if( fopen_s( &stream, "crt_fgetpos.txt", "rb" ) ) {
      perror( "Trouble opening file" );
      return -1;
   }

   // Read some data and then save the position.
   fread( buffer, sizeof( char ), 8, stream );
   if( fgetpos( stream, &pos ) != 0 ) {
      perror( "fgetpos error" );
      return -1;
   }

   fread( buffer, sizeof( char ), 13, stream );
   printf( "after fgetpos: %.13s\n", buffer );

   // Restore to old position and read data
   if( fsetpos( stream, &pos ) != 0 ) {
      perror( "fsetpos error" );
      return -1;
   }

   fread( buffer, sizeof( char ), 13, stream );
   printf( "after fsetpos: %.13s\n", buffer );
   fclose( stream );
}
```

Input: crt_fgetpos.txt

```
fgetpos gets a stream's file-position indicator.
```

Output crt_fgetpos.txt

```
after fgetpos: gets a stream
after fsetpos: gets a stream
```

See also

Stream I/O
fsetpos

fgets, fgetws

10/31/2018 • 2 minutes to read • Edit Online

Get a string from a stream.

Syntax

```
char *fgets(
   char *str,
   int numChars,
   FILE *stream
);
wchar_t *fgetws(
   wchar_t *str,
   int numChars,
   FILE *stream
);
```

Parameters

str
Storage location for data.

numChars
Maximum number of characters to read.

stream
Pointer to **FILE** structure.

Return Value

Each of these functions returns *str*. **NULL** is returned to indicate an error or an end-of-file condition. Use **feof** or **ferror** to determine whether an error occurred. If *str* or *stream* is a null pointer, or *numChars* is less than or equal to zero, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **fgets** function reads a string from the input *stream* argument and stores it in *str*. **fgets** reads characters from the current stream position to and including the first newline character, to the end of the stream, or until the number of characters read is equal to *numChars* - 1, whichever comes first. The result stored in *str* is appended with a null character. The newline character, if read, is included in the string.

fgetws is a wide-character version of **fgets**.

fgetws reads the wide-character argument *str* as a multibyte-character string or a wide-character string according to whether *stream* is opened in text mode or binary mode, respectively. For more information about using text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fgetts	**fgets**	**fgets**	**fgetws**

Requirements

FUNCTION	REQUIRED HEADER
fgets	<stdio.h>
fgetws	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fgets.c
// This program uses fgets to display
// the first line from a file.

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char line[100];

   if( fopen_s( &stream, "crt_fgets.txt", "r" ) == 0 )
   {
      if( fgets( line, 100, stream ) == NULL)
         printf( "fgets error\numChars" );
      else
         printf( "%s", line);
      fclose( stream );
   }
}
```

Input: crt_fgets.txt

```
Line one.
Line two.
```

Output

```
Line one.
```

See also

Stream I/O
fputs, fputws
gets, _getws
puts, _putws

filelength

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _filelength instead.

_filelength, _filelengthi64

10/31/2018 • 2 minutes to read • Edit Online

Gets the length of a file.

Syntax

```
long _filelength(
   int fd
);
__int64 _filelengthi64(
   int fd
);
```

Parameters

fd
Target the file descriptor.

Return Value

Both **_filelength** and **_filelengthi64** return the file length, in bytes, of the target file associated with *fd*. If *fd* is an invalid file descriptor, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, both functions return -1L to indicate an error and set **errno** to **EBADF**.

Requirements

FUNCTION	REQUIRED HEADER
_filelength	<io.h>
_filelengthi64	<io.h>

For more compatibility information, see Compatibility.

Example

See the example for _chsize.

See also

File Handling
_chsize
_fileno
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_stat, _wstat Functions
_stat, _wstat Functions

fileno

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fileno instead.

_fileno

10/31/2018 • 2 minutes to read • Edit Online

Gets the file descriptor associated with a stream.

Syntax

```
int _fileno(
   FILE *stream
);
```

Parameters

stream
Pointer to the **FILE** structure.

Return Value

_fileno returns the file descriptor. There is no error return. The result is undefined if *stream* does not specify an open file. If stream is **NULL**, **_fileno** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

> **NOTE**
> If **stdout** or **stderr** is not associated with an output stream (for example, in a Windows application without a console window), the file descriptor returned is -2. In previous versions, the file descriptor returned was -1. This change allows applications to distinguish this condition from an error.

Remarks

The **_fileno** routine returns the file descriptor currently associated with *stream*. This routine is implemented both as a function and as a macro. For information about choosing either implementation, see Choosing Between Functions and Macros.

Requirements

FUNCTION	REQUIRED HEADER
_fileno	<stdio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fileno.c
// This program uses _fileno to obtain
// the file descriptor for some standard C streams.
//

#include <stdio.h>

int main( void )
{
   printf( "The file descriptor for stdin is %d\n", _fileno( stdin ) );
   printf( "The file descriptor for stdout is %d\n", _fileno( stdout ) );
   printf( "The file descriptor for stderr is %d\n", _fileno( stderr ) );
}
```

```
The file descriptor for stdin is 0
The file descriptor for stdout is 1
The file descriptor for stderr is 2
```

See also

Stream I/O
_fdopen, _wfdopen
_filelength, _filelengthi64
fopen, _wfopen
freopen, _wfreopen

_findclose

10/31/2018 • 2 minutes to read • Edit Online

Closes the specified search handle and releases associated resources.

Syntax

```
int _findclose(
   intptr_t handle
);
```

Parameters

handle
Search handle returned by a previous call to **_findfirst**.

Return Value

If successful, **_findclose** returns 0. Otherwise, it returns -1 and sets **errno** to **ENOENT**, indicating that no more matching files could be found.

Requirements

FUNCTION	REQUIRED HEADER
_findclose	<io.h>

For more compatibility information, see Compatibility.

See also

System Calls
Filename Search Functions

_findfirst, _findfirst32, _findfirst32i64, _findfirst64, _findfirst64i32, _findfirsti64, _wfindfirst, _wfindfirst32, _wfindfirst32i64, _wfindfirst64, _wfindfirst64i32, _wfindfirsti64

2/4/2019 • 3 minutes to read • Edit Online

Provide information about the first instance of a file name that matches the file specified in the *filespec* argument.

Syntax

```
intptr_t _findfirst(
   const char *filespec,
   struct _finddata_t *fileinfo
);
intptr_t _findfirst32(
   const char *filespec,
   struct _finddata32_t *fileinfo
);
intptr_t _findfirst64(
   const char *filespec,
   struct _finddata64_t *fileinfo
);
intptr_t _findfirsti64(
   const char *filespec,
   struct _finddatai64_t *fileinfo
);
intptr_t _findfirst32i64(
   const char *filespec,
   struct _finddata32i64_t *fileinfo
);
intptr_t _findfirst64i32(
   const char *filespec,
   struct _finddata64i32_t *fileinfo
);
intptr_t _wfindfirst(
   const wchar_t *filespec,
   struct _wfinddata_t *fileinfo
);
intptr_t _wfindfirst32(
   const wchar_t *filespec,
   struct _wfinddata32_t *fileinfo
);
intptr_t _wfindfirst64(
   const wchar_t *filespec,
   struct _wfinddata64_t *fileinfo
);
intptr_t _wfindfirsti64(
   const wchar_t *filespec,
   struct _wfinddatai64_t *fileinfo
);
intptr_t _wfindfirst32i64(
   const wchar_t *filespec,
   struct _wfinddata32i64_t *fileinfo
);
intptr_t _wfindfirst64i32(
   const wchar_t *filespec,
   struct _wfinddata64i32_t *fileinfo
);
```

Parameters

filespec
Target file specification (can include wildcard characters).

fileinfo
File information buffer.

Return Value

If successful, **_findfirst** returns a unique search handle identifying the file or group of files that match the *filespec* specification, which can be used in a subsequent call to _findnext or to _findclose. Otherwise, **_findfirst** returns -1 and sets **errno** to one of the following values.

ERRNO VALUE	CONDITION
EINVAL	Invalid parameter: *filespec* or *fileinfo* was **NULL**. Or, the operating system returned an unexpected error.
ENOENT	File specification that could not be matched.
ENOMEM	Insufficient memory.
EINVAL	Invalid file name specification or the file name given was larger than **MAX_PATH**.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

Remarks

You must call _findclose after you are finished with either the **_findfirst** or _findnext function (or any variants). This frees resources used by these functions in your application.

The variations of these functions that have the **w** prefix are wide-character versions; otherwise, they are identical to the corresponding single-byte functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file sizes. The first numeric suffix (**32** or **64**) indicates the size of the time type; the second suffix is either **i32** or **i64**, and indicates whether the file size is represented as a 32-bit or 64-bit integer. For information about which versions support 32-bit and 64-bit time types and file sizes, see the following table. The **i32** or **i64** suffix is omitted if it is the same as the size of the time type, so **_findfirst64** also supports 64-bit file lengths and **_findfirst32** supports only 32-bit file lengths.

These functions use various forms of the **_finddata_t** structure for the *fileinfo* parameter. For more information about the structure, see Filename Search Functions.

The variations that use a 64-bit time type enable file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC. Those that use 32-bit time types represent dates only through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Unless you have a specific reason to use the versions that specify the time size explicitly, use **_findfirst** or **_wfindfirst** or, if you need to support file sizes larger than 3 GB, use **_findfirsti64** or **_wfindfirsti64**. All these functions use the 64-bit time type. In earlier versions, these functions used a 32-bit time type. If this is a breaking change for an application, you might define **_USE_32BIT_TIME_T** to revert to the old behavior. If **_USE_32BIT_TIME_T** is defined, **_findfirst**, **_finfirsti64**, and their corresponding Unicode versions use a 32-bit time.

Time Type and File Length Type Variations of _findfirst

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_findfirst, **_wfindfirst**	Not defined	64-bit	32-bit
_findfirst, **_wfindfirst**	Defined	32-bit	32-bit
_findfirst32, **_wfindfirst32**	Not affected by the macro definition	32-bit	32-bit

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_findfirst64, **_wfindfirst64**	Not affected by the macro definition	64-bit	64-bit
_findfirsti64, **_wfindfirsti64**	Not defined	64-bit	64-bit
_findfirsti64, **_wfindfirsti64**	Defined	32-bit	64-bit
_findfirst32i64, **_wfindfirst32i64**	Not affected by the macro definition	32-bit	64-bit
_findfirst64i32, **_wfindfirst64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfindfirst	**_findfirst**	**_findfirst**	**_wfindfirst**
_tfindfirst32	**_findfirst32**	**_findfirst32**	**_wfindfirst32**
_tfindfirst64	**_findfirst64**	**_findfirst64**	**_wfindfirst64**
_tfindfirsti64	**_findfirsti64**	**_findfirsti64**	**_wfindfirsti64**
_tfindfirst32i64	**_findfirst32i64**	**_findfirst32i64**	**_wfindfirst32i64**
_tfindfirst64i32	**_findfirst64i32**	**_findfirst64i32**	**_wfindfirst64i32**

Requirements

FUNCTION	REQUIRED HEADER
_findfirst	<io.h>
_findfirst32	<io.h>
_findfirst64	<io.h>
_findfirsti64	<io.h>
_findfirst32i64	<io.h>
_findfirst64i32	<io.h>
_wfindfirst	<io.h> or <wchar.h>
_wfindfirst32	<io.h> or <wchar.h>

FUNCTION	REQUIRED HEADER
_wfindfirst64	<io.h> or <wchar.h>
_wfindfirsti64	<io.h> or <wchar.h>
_wfindfirst32i64	<io.h> or <wchar.h>
_wfindfirst64i32	<io.h> or <wchar.h>

For more compatibility information, see Compatibility.

See also

System Calls
Filename Search Functions

_findnext, _findnext32, _findnext32i64, _findnext64, _findnext64i32, _findnexti64, _wfindnext, _wfindnext32, _wfindnext32i64, _wfindnext64, _wfindnext64i32, _wfindnexti64

11/8/2018 • 3 minutes to read • Edit Online

Find the next name, if any, that matches the *filespec* argument in a previous call to _findfirst, and then alter the *fileinfo* structure contents accordingly.

Syntax

```
int _findnext(
   intptr_t handle,
   struct _finddata_t *fileinfo
);
int _findnext32(
   intptr_t handle,
   struct _finddata32_t *fileinfo
);
int _findnext64(
   intptr_t handle,
   struct __finddata64_t *fileinfo
);
int _findnexti64(
   intptr_t handle,
   struct __finddatai64_t *fileinfo
);
int _findnext32i64(
   intptr_t handle,
   struct _finddata32i64_t *fileinfo
);
int _findnext64i32(
   intptr_t handle,
   struct _finddata64i32_t *fileinfo
);
int _wfindnext(
   intptr_t handle,
   struct _wfinddata_t *fileinfo
);
int _wfindnext32(
   intptr_t handle,
   struct _wfinddata32_t *fileinfo
);
int _wfindnext64(
   intptr_t handle,
   struct _wfinddata64_t *fileinfo
);
int _wfindnexti64(
   intptr_t handle,
   struct _wfinddatai64_t *fileinfo
);
int _wfindnext32i64(
   intptr_t handle,
   struct _wfinddatai64_t *fileinfo
);
int _wfindnext64i32(
   intptr_t handle,
   struct _wfinddata64i32_t *fileinfo
);
```

Parameters

handle
Search handle returned by a previous call to **_findfirst**.

fileinfo
File information buffer.

Return Value

If successful, returns 0. Otherwise, returns -1 and sets **errno** to a value indicating the nature of the failure. Possible error codes are shown in the following table.

ERRNO VALUE	CONDITION
EINVAL	Invalid parameter: *fileinfo* was **NULL**. Or, the operating system returned an unexpected error.
ENOENT	No more matching files could be found.
ENOMEM	Not enough memory or the file name's length exceeded **MAX_PATH**.

If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

Remarks

You must call _findclose after you are finished using either the **_findfirst** or **_findnext** function (or any variants). This frees up resources used by these functions in your application.

The variations of these functions with the **w** prefix are wide-character versions; otherwise, they are identical to the corresponding single-byte functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file sizes. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer. For information about which versions support 32-bit and 64-bit time types and file sizes, see the following table. The variations that use a 64-bit time type allow file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas those using 32-bit time types only represent dates through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Unless you have a specific reason to use the versions that specify the time size explicitly, use **_findnext** or **_wfindnext** or, if you need to support file sizes greater than 3 GB, use **_findnexti64** or **_wfindnexti64**. All these functions use the 64-bit time type. In previous versions, these functions used a 32-bit time type. If this is a breaking change for an application, you might define **_USE_32BIT_TIME_T** to get the old behavior. If **_USE_32BIT_TIME_T** is defined, **_findnext**, **_finnexti64** and their corresponding Unicode versions use a 32-bit time.

Time Type and File Length Type Variations of _findnext

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_findnext, **_wfindnext**	Not defined	64-bit	32-bit
_findnext, **_wfindnext**	Defined	32-bit	32-bit
_findnext32, **_wfindnext32**	Not affected by the macro definition	32-bit	32-bit
_findnext64, **_wfindnext64**	Not affected by the macro definition	64-bit	64-bit
_findnexti64, **_wfindnexti64**	Not defined	64-bit	64-bit
_findnexti64, **_wfindnexti64**	Defined	32-bit	64-bit

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_findnext32i64, **_wfindnext32i64**	Not affected by the macro definition	32-bit	64-bit
_findnext64i32, **_wfindnext64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfindnext	**_findnext**	**_findnext**	**_wfindnext**
_tfindnext32	**_findnext32**	**_findnext32**	**_wfindnext32**
_tfindnext64	**_findnext64**	**_findnext64**	**_wfindnext64**
_tfindnexti64	**_findnexti64**	**_findnexti64**	**_wfindnexti64**
_tfindnext32i64	**_findnext32i64**	**_findnext32i64**	**_wfindnext32i64**
_tfindnext64i32	**_findnext64i32**	**_findnext64i32**	**_wfindnext64i32**

Requirements

FUNCTION	REQUIRED HEADER
_findnext	<io.h>
_findnext32	<io.h>
_findnext64	<io.h>
_findnexti64	<io.h>
_findnext32i64	<io.h>
_findnext64i32	<io.h>
_wfindnext	<io.h> or <wchar.h>
_wfindnext32	<io.h> or <wchar.h>
_wfindnext64	<io.h> or <wchar.h>
_wfindnexti64	<io.h> or <wchar.h>
_wfindnext32i64	<io.h> or <wchar.h>
_wfindnext64i32	<io.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

System Calls
Filename Search Functions

Floating-point primitives

2/4/2019 • 6 minutes to read • Edit Online

Microsoft-specific primitive functions that are used to implement some standard C runtime library (CRT) floating-point functions. They're documented here for completeness, but aren't recommended for use. Some of these functions are noted as unused, because they're known to have issues in precision, exception handling, and conformance to IEEE-754 behavior. They exist in the library only for backward compatibility. For correct behavior, portability, and adherence to standards, prefer the standard floating-point functions over these functions.

_dclass, _ldclass, _fdclass

Syntax

```
short __cdecl _dclass(double x);
short __cdecl _ldclass(long double x);
short __cdecl _fdclass(float x);
```

Parameters

x
Floating-point function argument.

Remarks

These floating-point primitives implement the C versions of the CRT macro fpclassify for floating-point types. The classification of the argument *x* is returned as one of these constants, defined in math.h:

VALUE	DESCRIPTION
FP_NAN	A quiet, signaling, or indeterminate NaN
FP_INFINITE	A positive or negative infinity
FP_NORMAL	A positive or negative normalized non-zero value
FP_SUBNORMAL	A positive or negative subnormal (denormalized) value
FP_ZERO	A positive or negative zero value

For additional detail, you can use the Microsoft-specific _fpclass, _fpclassf functions. Use the fpclassify macro or function for portability.

_dsign, _ldsign, _fdsign

Syntax

```
int __cdecl _dsign(double x);
int __cdecl _ldsign(long double x);
int __cdecl _fdsign(float x);
```

Parameters

x
Floating-point function argument.

Remarks

These floating-point primitives implement the signbit macro or function in the CRT. They return a non-zero value if the sign bit is set in the significand (mantissa) of the argument *x*, and 0 if the sign bit is not set.

_dpcomp, _ldpcomp, _fdpcomp

Syntax

```
int __cdecl _dpcomp(double x, double y);
int __cdecl _ldpcomp(long double x, long double y);
int __cdecl _fdpcomp(float x, float y);
```

Parameters

x, *y*
Floating-point function arguments.

Remarks

These floating-point primitives take two arguments, *x* and *y*, and return a value that shows their ordering relationship, expressed as the bitwise or of these constants, defined in math.h:

VALUE	DESCRIPTION
_FP_LT	*x* can be considered less than *y*
_FP_EQ	*x* can be considered equal to *y*
_FP_GT	*x* can be considered greater than *y*

These primitives implement the isgreater, isgreaterequal, isless, islessequal, islessgreater, and isunordered macros and functions in the CRT.

_dtest, _ldtest, _fdtest

Syntax

```
short __cdecl _dtest(double* px);
short __cdecl _ldtest(long double* px);
short __cdecl _fdtest(float* px);
```

Parameters

px
Pointer to a floating-point argument.

Remarks

These floating-point primitives implement the C++ versions of the CRT function fpclassify for floating-point types. The argument *x* is evaluated and the classification is returned as one of these constants, defined in math.h:

VALUE	DESCRIPTION
FP_NAN	A quiet, signaling, or indeterminate NaN

VALUE	DESCRIPTION
FP_INFINITE	A positive or negative infinity
FP_NORMAL	A positive or negative normalized non-zero value
FP_SUBNORMAL	A positive or negative subnormal (denormalized) value
FP_ZERO	A positive or negative zero value

For additional detail, you can use the Microsoft-specific _fpclass, _fpclassf functions. Use the fpclassify function for portability.

_d_int, _ld_int, _fd_int

Syntax

```
short __cdecl _d_int(double* px, short exp);
short __cdecl _ld_int(long double* px, short exp);
short __cdecl _fd_int(float* px, short exp);
```

Parameters

px
Pointer to a floating-point argument.

exp
An exponent as an integral type.

Remarks

These floating-point primitives take a pointer to a floating-point value *px* and an exponent value *exp*, and remove the fractional part of the floating-point value below the given exponent, if possible. The value returned is the result of **fpclassify** on the input value in *px* if it's a NaN or infinity, and on the output value in *px* otherwise.

_dscale, _ldscale, _fdscale

Syntax

```
short __cdecl _dscale(double* px, long exp);
short __cdecl _ldscale(long double* px, long exp);
short __cdecl _fdscale(float* px, long exp);
```

Parameters

px
Pointer to a floating-point argument.

exp
An exponent as an integral type.

Remarks

These floating-point primitives take a pointer to a floating-point value *px* and an exponent value *exp*, and scale the value in *px* by 2^{exp}, if possible. The value returned is the result of **fpclassify** on the input value in *px* if it's a NaN or infinity, and on the output value in *px* otherwise. For portability, prefer the ldexp, ldexpf, and ldexpl functions.

_dunscale, _ldunscale, _fdunscale

Syntax

```
short __cdecl _dunscale(short* pexp, double* px);
short __cdecl _ldunscale(short* pexp, long double* px);
short __cdecl _fdunscale(short* pexp, float* px);
```

Parameters

pexp
A pointer to an exponent as an integral type.

px
Pointer to a floating-point argument.

Remarks

These floating-point primitives break down the floating-point value pointed at by *px* into a significand (mantissa) and an exponent, if possible. The significand is scaled such that the absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is the value *n*, where the original floating-point value is equal to the scaled significand times 2^n. This integer exponent *n* is stored at the location pointed to by *pexp*. The value returned is the result of **fpclassify** on the input value in *px* if it's a NaN or infinity, and on the output value otherwise. For portability, prefer the frexp, frexpf, frexpl functions.

_dexp, _ldexp, _fdexp

Syntax

```
short __cdecl _dexp(double* px, double y, long exp);
short __cdecl _ldexp(long double* px, long double y, long exp);
short __cdecl _fdexp(float* px, float y, long exp);
```

Parameters

y
Floating-point function argument.

px
Pointer to a floating-point argument.

exp
An exponent as an integral type.

Remarks

These floating-point primitives construct a floating-point value in the location pointed at by *px* equal to $y * 2^{exp}$. The value returned is the result of **fpclassify** on the input value in *y* if it's a NaN or infinity, and on the output value in *px* otherwise. For portability, prefer the ldexp, ldexpf, and ldexpl functions.

_dnorm, _fdnorm

Syntax

```
short __cdecl _dnorm(unsigned short* ps);
short __cdecl _fdnorm(unsigned short* ps);
```

Parameters

ps
Pointer to the bitwise representation of a floating-point value expressed as an array of **unsigned short**.

Remarks

These floating-point primitives normalize the fractional part of an underflowed floating-point value and adjust the *characteristic*, or biased exponent, to match. The value is passed as the bitwise representation of the floating-point type converted to an array of **unsigned short** through the `_double_val`, `_ldouble_val`, or `_float_val` type punning union declared in math.h. The return value is the result of **fpclassify** on the input floating-point value if it's a NaN or infinity, and on the output value otherwise.

_dpoly, _ldpoly, _fdpoly

Syntax

```
double __cdecl _dpoly(double x, double const* table, int n);
long double __cdecl _ldpoly(long double x, long double const* table, int n);
float __cdecl _fdpoly(float x, _float const* table, int n);
```

Parameters

x
Floating-point function argument.

table
Pointer to a table of constant coefficients for a polynomial.

n
Order of the polynomial to evaluate.

Remarks

These floating-point primitives return the evaluation of *x* in the polynomial of order *n* whose coefficients are represented by the corresponding constant values in *table*. For example, if *table*[0] = 3.0, *table*[1] = 4.0, *table*[2] = 5.0, and *n* = 2, it represents the polynomial $5.0x^2 + 4.0x + 3.0$. If this polynomial is evaluated for *x* of 2.0, the result is 31.0. These functions aren't used internally.

_dlog, _dlog, _dlog

Syntax

```
double __cdecl _dlog(double x, int base_flag);
long double __cdecl _ldlog(long double x, int base_flag);
float __cdecl _fdlog(float x, int base_flag);
```

Parameters

x
Floating-point function argument.

base_flag
Flag that controls the base to use, 0 for base *e* and non-zero for base 10.

Remarks

These floating-point primitives return the natural log of *x*, ln(*x*) or $\log_e(x)$, when *base_flag* is 0. They return the log base 10 of *x*, or $\log_{10}(x)$, when *base_flag* is non-zero. These functions aren't used internally. For portability, prefer the functions log, logf, logl, log10, log10f, and log10l.

_dsin, _ldsin, _fdsin

Syntax

```
double __cdecl _dsin(double x, unsigned int quadrant);
long double __cdecl _ldsin(long double x, unsigned int quadrant);
float __cdecl _fdsin(float x, unsigned int quadrant);
```

Parameters

x
Floating-point function argument.

quadrant
Quadrant offset of 0, 1, 2, or 3 to use to produce `sin`, `cos`, `-sin`, and `-cos` results.

Remarks

These floating-point primitives return the sine of *x* offset by the *quadrant* modulo 4. Effectively, they return the sine, cosine, -sine, and -cosine of *x* when *quadrant* modulo 4 is 0, 1, 2, or 3, respectively. These functions aren't used internally. For portability, prefer the sin, sinf, sinl, cos, cosf, and cosl functions.

Requirements

Header: <math.h>

For additional compatibility information, see Compatibility.

See also

Floating-point support
fpclassify
_fpclass, _fpclassf
isfinite, _finite, _finitef
isinf
isnan, _isnan, _isnanf
isnormal
cos, cosf, cosl
frexp, frexpf, frexpl
ldexp, ldexpf, and ldexpl
log, logf, logl, log10, log10f, log10l
sin, sinf, sinl

floor, floorf, floorl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the floor of a value.

Syntax

```
double floor(
   double x
);
float floor(
   float x
); // C++ only
long double floor(
   long double x
); // C++ only
float floorf(
   float x
);
long double floorl(
   long double x
);
```

Parameters

x
Floating-point value.

Return Value

The **floor** functions return a floating-point value that represents the largest integer that is less than or equal to *x*. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	none	_DOMAIN

floor has an implementation that uses Streaming SIMD Extensions 2 (SSE2). For information and restrictions about using the SSE2 implementation, see _set_SSE2_enable.

Remarks

C++ allows overloading, so you can call overloads of **floor** that take and return **float** and **long double** values. In a C program, **floor** always takes and returns a **double**.

Requirements

FUNCTION	REQUIRED HEADER
floor, **floorf**, **floorl**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_floor.c
// This example displays the largest integers
// less than or equal to the floating-point values 2.8
// and -2.8. It then shows the smallest integers greater
// than or equal to 2.8 and -2.8.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double y;

   y = floor( 2.8 );
   printf( "The floor of 2.8 is %f\n", y );
   y = floor( -2.8 );
   printf( "The floor of -2.8 is %f\n", y );

   y = ceil( 2.8 );
   printf( "The ceil of 2.8 is %f\n", y );
   y = ceil( -2.8 );
   printf( "The ceil of -2.8 is %f\n", y );
}
```

```
The floor of 2.8 is 2.000000
The floor of -2.8 is -3.000000
The ceil of 2.8 is 3.000000
The ceil of -2.8 is -2.000000
```

See also

Floating-Point Support
ceil, ceilf, ceill
round, roundf, roundl
fmod, fmodf

flushall

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _flushall instead.

_flushall

10/31/2018 • 2 minutes to read • Edit Online

Flushes all streams; clears all buffers.

Syntax

```
int _flushall( void );
```

Return Value

_flushall returns the number of open streams (input and output). There is no error return.

Remarks

By default, the **_flushall** function writes to appropriate files the contents of all buffers associated with open output streams. All buffers associated with open input streams are cleared of their current contents. (These buffers are normally maintained by the operating system, which determines the optimal time to write the data automatically to disk: when a buffer is full, when a stream is closed, or when a program terminates normally without closing streams.)

If a read follows a call to **_flushall**, new data is read from the input files into the buffers. All streams remain open after the call to **_flushall**.

The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating system buffers. Without rewriting an existing program, you can enable this feature by linking the program's object files with Commode.obj. In the resulting executable file, calls to **_flushall** write the contents of all buffers to disk. Only **_flushall** and fflush are affected by Commode.obj.

For information about controlling the commit-to-disk feature, see Stream I/O, fopen, and _fdopen.

Requirements

FUNCTION	REQUIRED HEADER
_flushall	<stdio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_flushall.c
// This program uses _flushall
// to flush all open buffers.

#include <stdio.h>

int main( void )
{
   int numflushed;

   numflushed = _flushall();
   printf( "There were %d streams flushed\n", numflushed );
}
```

```
There were 3 streams flushed
```

See also

Stream I/O
_commit
fclose, _fcloseall
fflush
setvbuf

fma, fmaf, fmal

11/9/2018 • 2 minutes to read • Edit Online

Multiplies two values together, adds a third value, and then rounds the result, without losing any precision due to intermediary rounding.

Syntax

```
double fma(
   double x,
   double y,
   double z
);

float fma(
   float  x,
   float  y,
   float z
); //C++ only

long double fma(
   long double  x,
   long double  y,
   long double z
); //C++ only

float fmaf(
   float  x,
   float  y,
   float z
);

long double fmal(
   long double  x,
   long double  y,
   long double z
);
```

Parameters

x
The first value to multiply.

y
The second value to multiply.

z
The value to add.

Return Value

Returns `(x * y) + z`. The return value is then rounded using the current rounding format.

Otherwise, may return one of the following values:

ISSUE	RETURN
x = INFINITY, *y* = 0 or *x* = 0, *y* = INFINITY	NaN
x or *y* = exact ± INFINITY, *z* = INFINITY with the opposite sign	NaN
x or *y* = NaN	NaN
not (*x* = 0, *y*= indefinite) and *z* = NaN not (*x*=indefinite, *y*=0) and *z* = NaN	NaN
Overflow range error	±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL
Underflow range error	correct value, after rounding.

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **fma** that take and return **float** and **long double** types. In a C program, **fma** always takes and returns a **double**.

This function computes the value as though it were taken to infinite precision, and then rounds the final result.

Requirements

FUNCTION	C HEADER	C++ HEADER
fma, **fmaf**, **fmal**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
remainder, remainderf, remainderl
remquo, remquof, remquol

fmax, fmaxf, fmaxl

11/9/2018 • 2 minutes to read • Edit Online

Determine the larger of two specified numeric values.

Syntax

```
double fmax(
   double x,
   double y
);

float fmax(
   float x,
   float y
); //C++ only

long double fmax(
   long double x,
   long double y
); //C++ only

float fmaxf(
   float x,
   float y
);

long double fmaxl(
   long double x,
   long double y
);
```

Parameters

x
The first value to compare.

y
The second value to compare.

Return Value

If successful, returns the larger of *x* or *y*. The value returned is exact, and does not depend on any form of rounding.

Otherwise, may return one of the following values:

ISSUE	RETURN
x = NaN	*y*
y = NaN	*x*
x and *y* = NaN	NaN

This function does not use the errors specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of fmax that take and return float and long double types. In a C program, fmax always takes and returns a double.

Requirements

FUNCTION	C HEADER	C++ HEADER
fmax, **fmaxf**, **fmaxl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fmin, fminf, fminl

fmin, fminf, fminl

10/31/2018 • 2 minutes to read • Edit Online

Determines the smaller of the two specified values.

Syntax

```
double fmin(
   double x,
   double y
);

float fmin(
   float x,
   float y
); //C++ only

long double fmin(
   long double x,
   long double y
); //C++ only

float fminf(
   float x,
   float y
);

long double fminl(
   long double x,
   long double y
);
```

Parameters

x
The first value to compare.

y
The second value to compare.

Return Value

If successful, returns the smaller of *x* or *y*.

INPUT	RESULT
x is NaN	*y*
y is NaN	*x*
x and *y* are NaN	NaN

The function does not cause _matherr to be invoked, cause any floating-point exceptions, or change the value of **errno**.

Remarks

Because C++ allows overloading, you can call overloads of **fmin** that take and return **float** and **long double** types. In a C program, **fmin** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER
fmin, **fminf**, **fminl**	C: <math.h> C++: <math.h> or <cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
fmax, fmaxf, fmaxl

fmod, fmodf, fmodl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the floating-point remainder.

Syntax

```
double fmod(
   double x,
   double y
);
float fmod(
   float x,
   float y
);  // C++ only
long double fmod(
   long double x,
   long double y
);  // C++ only
float fmodf(
   float x,
   float y
);
long double fmodl(
   long double x,
   long double y
);
```

Parameters

x, *y*
Floating-point values.

Return Value

fmod returns the floating-point remainder of *x* / *y*. If the value of *y* is 0.0, **fmod** returns a quiet NaN. For information about representation of a quiet NaN by the **printf** family, see printf.

Remarks

The **fmod** function calculates the floating-point remainder *f* of *x* / *y* such that *x* = *i* * *y* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.

C++ allows overloading, so you can call overloads of **fmod** that take and return **float** and **long double** values. In a C program, **fmod** always takes two **double** arguments and returns a **double**.

Requirements

FUNCTION	REQUIRED HEADER
fmod, **fmodf**, **fmodl**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fmod.c
// This program displays a floating-point remainder.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double w = -10.0, x = 3.0, z;

   z = fmod( w, x );
   printf( "The remainder of %.2f / %.2f is %f\n", w, x, z );
}
```

```
The remainder of -10.00 / 3.00 is -1.000000
```

See also

Floating-Point Support
ceil, ceilf, ceill
fabs, fabsf, fabsl
floor, floorf, floorl
_CIfmod

fopen, _wfopen

2/4/2019 • 11 minutes to read • Edit Online

Opens a file. More-secure versions of these functions that perform additional parameter validation and return error codes are available; see fopen_s, _wfopen_s.

Syntax

```
FILE *fopen(
   const char *filename,
   const char *mode
);
FILE *_wfopen(
   const wchar_t *filename,
   const wchar_t *mode
);
```

Parameters

filename
File name.

mode
Kind of access that's enabled.

Return Value

Each of these functions returns a pointer to the open file. A null pointer value indicates an error. If *filename* or *mode* is **NULL** or an empty string, these functions trigger the invalid parameter handler, which is described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **fopen** function opens the file that is specified by *filename*. By default, a narrow *filename* string is interpreted using the ANSI codepage (CP_ACP). In Windows Desktop applications this can be changed to the OEM codepage (CP_OEMCP) by using the SetFileApisToOEM function. You can use the AreFileApisANSI function to determine whether *filename* is interpreted using the ANSI or the system default OEM codepage. **_wfopen** is a wide-character version of **fopen**; the arguments to **_wfopen** are wide-character strings. Otherwise, **_wfopen** and **fopen** behave identically. Just using **_wfopen** does not affect the coded character set that is used in the file stream.

fopen accepts paths that are valid on the file system at the point of execution; **fopen** accepts UNC paths and paths that involve mapped network drives as long as the system that executes the code has access to the share or mapped drive at the time of execution. When you construct paths for **fopen**, make sure that drives, paths, or network shares will be available in the execution environment. You can use either forward slashes (/) or backslashes (\) as the directory separators in a path.

Always check the return value to see whether the pointer is NULL before you perform any additional operations on the file. If an error occurs, the global variable **errno** is set and may be used to obtain

specific error information. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Unicode Support

fopen supports Unicode file streams. To open a Unicode file, pass a **ccs** flag that specifies the desired encoding to **fopen**, as follows.

```
FILE *fp = fopen("newfile.txt", "rt+, ccs=encoding");
```

Allowed values of *encoding* are **UNICODE**, **UTF-8**, and **UTF-16LE**.

When a file is opened in Unicode mode, input functions translate the data that's read from the file into UTF-16 data stored as type **wchar_t**. Functions that write to a file opened in Unicode mode expect buffers that contain UTF-16 data stored as type **wchar_t**. If the file is encoded as UTF-8, then UTF-16 data is translated into UTF-8 when it is written, and the file's UTF-8-encoded content is translated into UTF-16 when it is read. An attempt to read or write an odd number of bytes in Unicode mode causes a parameter validation error. To read or write data that's stored in your program as UTF-8, use a text or binary file mode instead of a Unicode mode. You are responsible for any required encoding translation.

If the file already exists and is opened for reading or appending, the Byte Order Mark (BOM), if it present in the file, determines the encoding. The BOM encoding takes precedence over the encoding that is specified by the **ccs** flag. The **ccs** encoding is only used when no BOM is present or the file is a new file.

> **NOTE**
> BOM detection only applies to files that are opened in Unicode mode (that is, by passing the **ccs** flag).

The following table summarizes the modes that are used for various **ccs** flags given to **fopen** and Byte Order Marks in the file.

Encodings Used Based on ccs Flag and BOM

CCS FLAG	NO BOM (OR NEW FILE)	BOM: UTF-8	BOM: UTF-16
UNICODE	**UTF-16LE**	**UTF-8**	**UTF-16LE**
UTF-8	**UTF-8**	**UTF-8**	**UTF-16LE**
UTF-16LE	**UTF-16LE**	**UTF-8**	**UTF-16LE**

Files opened for writing in Unicode mode have a BOM written to them automatically.

If *mode* is **"a, ccs=***encoding***"**, **fopen** first tries to open the file by using both read and write access. If this succeeds, the function reads the BOM to determine the encoding for the file; if this fails, the function uses the default encoding for the file. In either case, **fopen** will then re-open the file by using write-only access. (This applies to **"a"** mode only, not to **"a+"** mode.)

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfopen	**fopen**	**fopen**	**_wfopen**

The character string *mode* specifies the kind of access that is requested for the file, as follows.

MODE	ACCESS
"r"	Opens for reading. If the file does not exist or cannot be found, the **fopen** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending) without removing the end-of-file (EOF) marker before new data is written to the file. Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. The appending operation includes the removal of the EOF marker before new data is written to the file. The EOF marker is not restored after writing is completed. Creates the file if it does not exist.

When a file is opened by using the **"a"** access type or the **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned by using fseek or rewind, but is always moved back to the end of the file before any write operation is performed. Therefore, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before it appends to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. Before it appends to the file, the **"a+"** mode does remove the EOF marker. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are enabled (the file is said to be open for "update"). However, when you switch from reading to writing, the input operation must encounter an EOF marker. If there is no EOF, you must use an intervening call to a file positioning function. The file positioning functions are **fsetpos**, fseek, and rewind. When you switch from writing to reading, you must use an intervening call to either **fflush** or to a file positioning function.

In addition to the earlier values, the following characters can be appended to *mode* to specify the translation mode for newline characters.

MODE MODIFIER	TRANSLATION MODE
t	Open in text (translated) mode.
b	Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

In text mode, CTRL+Z is interpreted as an EOF character on input. In files that are opened for reading/writing by using **"a+"**, **fopen** checks for a CTRL+Z at the end of the file and removes it, if it is possible. This is done because using fseek and **ftell** to move within a file that ends with CTRL+Z may cause fseek to behave incorrectly near the end of the file.

In text mode, carriage return-linefeed combinations are translated into single linefeeds on input, and linefeed characters are translated to carriage return-linefeed combinations on output. When a Unicode stream-I/O function operates in text mode (the default), the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For more information about how to use text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

The following options can be appended to *mode* to specify additional behaviors.

***MODE* MODIFIER**	**BEHAVIOR**
c	Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.
n	Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with COMMODE.OBJ. The global commit flag default is "no-commit" unless you explicitly link your program with COMMODE.OBJ (see Link Options).
N	Specifies that the file is not inherited by child processes.
S	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
R	Specifies that caching is optimized for, but not restricted to, random access from disk.
T	Specifies a file as temporary. If possible, it is not flushed to disk.
D	Specifies a file as temporary. It is deleted when the last file pointer is closed.
ccs=*encoding*	Specifies the encoded character set to use (one of **UTF-8**, **UTF-16LE**, or **UNICODE**) for this file. Leave unspecified if you want ANSI encoding.

Valid characters for the *mode* string that is used in **fopen** and **_fdopen** correspond to *oflag* arguments that are used in _open and _sopen, as follows.

CHARACTERS IN *MODE* STRING	**EQUIVALENT *OFLAG* VALUE FOR _OPEN/_SOPEN**
a	**_O_WRONLY** | **_O_APPEND** (usually **_O_WRONLY** | **_O_CREAT** | **_O_APPEND**)

CHARACTERS IN *MODE* STRING	EQUIVALENT *OFLAG* VALUE FOR _OPEN/_SOPEN
a+	**_O_RDWR** | **_O_APPEND** (usually **_O_RDWR** | **_O_APPEND** | **_O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**
w	**_O_WRONLY** (usually **_O_WRONLY** | **_O_CREAT** | **_O_TRUNC**)
w+	**_O_RDWR** (usually **_O_RDWR** | **_O_CREAT** | **_O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None
S	**_O_SEQUENTIAL**
R	**_O_RANDOM**
T	**_O_SHORTLIVED**
D	**_O_TEMPORARY**
ccs=UNICODE	**_O_WTEXT**
ccs=UTF-8	**_O_UTF8**
ccs=UTF-16LE	**_O_UTF16**

If you are using **rb** mode, you do not have to port your code, and if you expect to read most of a large file or are not concerned about network performance, you might also consider whether to use memory mapped Win32 files as an option.

Requirements

FUNCTION	REQUIRED HEADER
fopen	<stdio.h>
_wfopen	<stdio.h> or <wchar.h>

_wfopen is a Microsoft extension. For more information about compatibility, see Compatibility.

The **c**, **n**, **t**, **S**, **R**, **T**, and **D** *mode* options are Microsoft extensions for **fopen** and **_fdopen** and should not be used where ANSI portability is desired.

Example 1

The following program opens two files. It uses **fclose** to close the first file and **_fcloseall** to close all remaining files.

```
// crt_fopen.c
// compile with: /W3
// This program opens two files. It uses
// fclose to close the first file and
// _fcloseall to close all remaining files.

#include <stdio.h>

FILE *stream, *stream2;

int main( void )
{
   int numclosed;

   // Open for read (will fail if file "crt_fopen.c" does not exist)
   if( (stream  = fopen( "crt_fopen.c", "r" )) == NULL ) // C4996
   // Note: fopen is deprecated; consider using fopen_s instead
      printf( "The file 'crt_fopen.c' was not opened\n" );
   else
      printf( "The file 'crt_fopen.c' was opened\n" );

   // Open for write
   if( (stream2 = fopen( "data2", "w+" )) == NULL ) // C4996
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );

   // Close stream if it is not NULL
   if( stream)
   {
      if ( fclose( stream ) )
      {
         printf( "The file 'crt_fopen.c' was not closed\n" );
      }
   }

   // All other files are closed:
   numclosed = _fcloseall( );
   printf( "Number of files closed by _fcloseall: %u\n", numclosed );
}
```

```
The file 'crt_fopen.c' was opened
The file 'data2' was opened
Number of files closed by _fcloseall: 1
```

Example 2

The following program creates a file (or overwrites one if it exists), in text mode that has Unicode encoding. It then writes two strings into the file and closes the file. The output is a file named _wfopen_test.xml, which contains the data from the output section.

```
// crt__wfopen.c
// compile with: /W3
// This program creates a file (or overwrites one if
// it exists), in text mode using Unicode encoding.
// It then writes two strings into the file
// and then closes the file.

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <wchar.h>

#define BUFFER_SIZE 50

int main(int argc, char** argv)
{
    wchar_t str[BUFFER_SIZE];
    size_t  strSize;
    FILE*   fileHandle;

    // Create an the xml file in text and Unicode encoding mode.
    if ((fileHandle = _wfopen( L"_wfopen_test.xml",L"wt+,ccs=UNICODE")) == NULL) // C4996
    // Note: _wfopen is deprecated; consider using _wfopen_s instead
    {
        wprintf(L"_wfopen failed!\n");
        return(0);
    }

    // Write a string into the file.
    wcscpy_s(str, sizeof(str)/sizeof(wchar_t), L"<xmlTag>\n");
    strSize = wcslen(str);
    if (fwrite(str, sizeof(wchar_t), strSize, fileHandle) != strSize)
    {
        wprintf(L"fwrite failed!\n");
    }

    // Write a string into the file.
    wcscpy_s(str, sizeof(str)/sizeof(wchar_t), L"</xmlTag>");
    strSize = wcslen(str);
    if (fwrite(str, sizeof(wchar_t), strSize, fileHandle) != strSize)
    {
        wprintf(L"fwrite failed!\n");
    }

    // Close the file.
    if (fclose(fileHandle))
    {
        wprintf(L"fclose failed!\n");
    }
    return 0;
}
```

See also

Stream I/O
Interpretation of Multibyte-Character Sequences
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
freopen, _wfreopen
_open, _wopen
_setmode

_sopen, _wsopen

fopen_s, _wfopen_s

11/8/2018 • 9 minutes to read • Edit Online

Opens a file. These versions of fopen, _wfopen have security enhancements, as described in Security Features in the CRT.

Syntax

```
errno_t fopen_s(
   FILE** pFile,
   const char *filename,
   const char *mode
);
errno_t _wfopen_s(
   FILE** pFile,
   const wchar_t *filename,
   const wchar_t *mode
);
```

Parameters

pFile
A pointer to the file pointer that will receive the pointer to the opened file.

filename
Filename.

mode
Type of access permitted.

Return Value

Zero if successful; an error code on failure. See errno, _doserrno, _sys_errlist, and _sys_nerr for more information about these error codes.

Error Conditions

PFILE	*FILENAME*	*MODE*	RETURN VALUE	CONTENTS OF *PFILE*
NULL	any	any	**EINVAL**	unchanged
any	**NULL**	any	**EINVAL**	unchanged
any	any	**NULL**	**EINVAL**	unchanged

Remarks

Files that are opened by **fopen_s** and **_wfopen_s** are not sharable. If you require that a file be sharable, use _fsopen, _wfsopen with the appropriate sharing mode constant—for example, **_SH_DENYNO** for read/write sharing.

The **fopen_s** function opens the file that's specified by *filename*. **_wfopen_s** is a wide-character version of **fopen_s**; the arguments to **_wfopen_s** are wide-character strings. **_wfopen_s** and **fopen_s** behave identically

otherwise.

fopen_s accepts paths that are valid on the file system at the point of execution; UNC paths and paths that involve mapped network drives are accepted by **fopen_s** as long as the system that's executing the code has access to the share or mapped network drive at the time of execution. When you construct paths for **fopen_s**, don't make assumptions about the availability of drives, paths, or network shares in the execution environment. You can use either forward slashes (/) or backslashes (\) as the directory separators in a path.

These functions validate their parameters. If *pFile*, *filename*, or *mode* is a null pointer, these functions generate an invalid parameter exception, as described in Parameter Validation.

Always check the return value to see if the function succeeded before you perform any further operations on the file. If an error occurs, the error code is returned and the global variable **errno** is set. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Unicode support

fopen_s supports Unicode file streams. To open a new or existing Unicode file, pass a *ccs* flag that specifies the desired encoding to **fopen_s**:

fopen_s(&fp, "newfile.txt", "rw, ccs=*encoding***");**

Allowed values of *encoding* are **UNICODE**, **UTF-8**, and **UTF-16LE**. If there no value is specified for *encoding*, **fopen_s** uses ANSI encoding.

If the file already exists and is opened for reading or appending, the Byte Order Mark (BOM), if present in the file, determines the encoding. The BOM encoding takes precedence over the encoding that's specified by the *ccs* flag. The *ccs* encoding is only used when no BOM is present or if the file is a new file.

> **NOTE**
> BOM-detection only applies to files that are opened in Unicode mode; that is, by passing the *ccs* flag.

The following table summarizes the modes for various *ccs* flags that are given to **fopen_s** and for Byte Order Marks in the file.

Encodings Used Based on ccs Flag and BOM

CCS FLAG	NO BOM (OR NEW FILE)	BOM: UTF-8	BOM: UTF-16
UNICODE	**UTF-16LE**	**UTF-8**	**UTF-16LE**
UTF-8	**UTF-8**	**UTF-8**	**UTF-16LE**
UTF-16LE	**UTF-16LE**	**UTF-8**	**UTF-16LE**

Files that are opened for writing in Unicode mode have a BOM written to them automatically.

If *mode* is **"a, ccs=***encoding***"**, **fopen_s** first tries to open the file with both read access and write access. If successful, the function reads the BOM to determine the encoding for the file; if unsuccessful, the function uses the default encoding for the file. In either case, **fopen_s** then re-opens the file with write-only access. (This applies to **a** mode only, not **a+**.)

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfopen_s	**fopen_s**	**fopen_s**	**_wfopen_s**

The character string *mode* specifies the kind of access that's requested for the file, as follows.

MODE	ACCESS
"r"	Opens for reading. If the file does not exist or cannot be found, the **fopen_s** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending) without removing the end-of-file (EOF) marker before new data is written to the file. Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. The appending operation includes the removal of the EOF marker before new data is written to the file. The EOF marker is not restored after writing is completed. Creates the file if it does not exist.

When a file is opened by using the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned by using fseek or rewind, but it's always moved back to the end of the file before any write operation is carried out so that existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data that's appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated by using the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed. (The file is said to be open for "update".) However, when you switch from reading to writing, the input operation must encounter an EOF marker. If there is no EOF, you must use an intervening call to a file-positioning function. The file-positioning functions are **fsetpos**, fseek, and rewind. When you switch from writing to reading, you must use an intervening call to either **fflush** or to a file-positioning function.

In addition to the above values, the following characters can be included in *mode* to specify the translation mode for newline characters:

MODE MODIFIER	TRANSLATION MODE
t	Open in text (translated) mode.
b	Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

In text (translated) mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for

reading/writing with **"a+"**, **fopen_s** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using fseek and **ftell** to move within a file that ends with a CTRL+Z, may cause fseek to behave improperly near the end of the file.

Also, in text mode, carriage return-linefeed combinations are translated into single linefeeds on input, and linefeed characters are translated to carriage return-linefeed combinations on output. When a Unicode stream-I/O function operates in text mode (the default), the source or destination stream is assumed to be a sequence of multibyte characters. Therefore, the Unicode stream-input functions convert multibyte characters to wide characters (as if by a call to the **mbtowc** function). For the same reason, the Unicode stream-output functions convert wide characters to multibyte characters (as if by a call to the **wctomb** function).

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For more information about using text and binary modes in Unicode and multibyte stream-I/O, see Text and Binary Mode File I/O and Unicode Stream I/O in Text and Binary Modes.

***MODE* MODIFIER**	**BEHAVIOR**
c	Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either **fflush** or **_flushall** is called.
n	Reset the commit flag for the associated *filename* to "no-commit." This is the default. It also overrides the global commit flag if you link your program with COMMODE.OBJ. The global commit flag default is "no-commit" unless you explicitly link your program with COMMODE.OBJ (see Link Options).
N	Specifies that the file is not inherited by child processes.
S	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
R	Specifies that caching is optimized for, but not restricted to, random access from disk.
T	Specifies a file as temporary. If possible, it is not flushed to disk.
D	Specifies a file as temporary. It is deleted when the last file pointer is closed.
ccs=*encoding*	Specifies the encoded character set to use (one of **UTF-8**, **UTF-16LE**, or **UNICODE**) for this file. Leave unspecified if you want ANSI encoding.

Valid characters for the *mode* string used in **fopen_s** and _fdopen correspond to *oflag* arguments used in _open and _sopen, as follows.

CHARACTERS IN *MODE* STRING	**EQUIVALENT *OFLAG* VALUE FOR _OPEN/_SOPEN**
a	**_O_WRONLY** | **_O_APPEND** (usually **_O_WRONLY** | **_O_CREAT** |** _O_APPEND**)

CHARACTERS IN *MODE* STRING	EQUIVALENT *OFLAG* VALUE FOR _OPEN/_SOPEN
a+	**_O_RDWR** | **_O_APPEND** (usually **_O_RDWR** | **_O_APPEND** | **_O_CREAT**)
r	**_O_RDONLY**
r+	**_O_RDWR**
w	**_O_WRONLY** (usually **_O_WRONLY** | **_O_CREAT** |** _O_TRUNC**)
w+	**_O_RDWR** (usually **_O_RDWR** | **_O_CREAT** | **_O_TRUNC**)
b	**_O_BINARY**
t	**_O_TEXT**
c	None
n	None
S	**_O_SEQUENTIAL**
R	**_O_RANDOM**
T	**_O_SHORTLIVED**
D	**_O_TEMPORARY**
ccs=UNICODE	**_O_WTEXT**
ccs=UTF-8	**_O_UTF8**
ccs=UTF-16LE	**_O_UTF16**

If you are using **rb** mode, won't need to port your code, and expect to read a lot of the file and/or don't care about network performance, memory mapped Win32 files might also be an option.

Requirements

FUNCTION	REQUIRED HEADER
fopen_s	<stdio.h>
_wfopen_s	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

The **c**, **n**, and **t** *mode* options are Microsoft extensions for **fopen_s** and _fdopen and should not be used where ANSI portability is desired.

Example

```
// crt_fopen_s.c
// This program opens two files. It uses
// fclose to close the first file and
// _fcloseall to close all remaining files.

#include <stdio.h>

FILE *stream, *stream2;

int main( void )
{
   errno_t err;

   // Open for read (will fail if file "crt_fopen_s.c" does not exist)
   err  = fopen_s( &stream, "crt_fopen_s.c", "r" );
   if( err == 0 )
   {
      printf( "The file 'crt_fopen_s.c' was opened\n" );
   }
   else
   {
      printf( "The file 'crt_fopen_s.c' was not opened\n" );
   }

   // Open for write
   err = fopen_s( &stream2, "data2", "w+" );
   if( err == 0 )
   {
      printf( "The file 'data2' was opened\n" );
   }
   else
   {
      printf( "The file 'data2' was not opened\n" );
   }

   // Close stream if it is not NULL
   if( stream )
   {
      err = fclose( stream );
      if ( err == 0 )
      {
         printf( "The file 'crt_fopen_s.c' was closed\n" );
      }
      else
      {
         printf( "The file 'crt_fopen_s.c' was not closed\n" );
      }
   }

   // All other files are closed:
   int numclosed = _fcloseall( );
   printf( "Number of files closed by _fcloseall: %u\n", numclosed );
}
```

```
The file 'crt_fopen_s.c' was opened
The file 'data2' was opened
Number of files closed by _fcloseall: 1
```

See also

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
freopen, _wfreopen
_open, _wopen
_setmode

_fpclass, _fpclassf

10/31/2018 • 2 minutes to read • Edit Online

Returns a value indicating the floating-point classification of the argument.

Syntax

```
int _fpclass(
   double x
);

int _fpclassf(
   float x
); /* x64 only */
```

Parameters

x
The floating-point value to test.

Return Value

The **_fpclass** and **_fpclassf** functions return an integer value that indicates the floating-point classification of the argument *x*. The classification may have one of the following values, defined in <float.h>.

VALUE	DESCRIPTION
_FPCLASS_SNAN	Signaling NaN
_FPCLASS_QNAN	Quiet NaN
_FPCLASS_NINF	Negative infinity (-INF)
_FPCLASS_NN	Negative normalized non-zero
_FPCLASS_ND	Negative denormalized
_FPCLASS_NZ	Negative zero (- 0)
_FPCLASS_PZ	Positive 0 (+0)
_FPCLASS_PD	Positive denormalized
_FPCLASS_PN	Positive normalized non-zero
_FPCLASS_PINF	Positive infinity (+INF)

Remarks

The **_fpclass** and **_fpclassf** functions are Microsoft specific. They are similar to fpclassify, but return more

detailed information about the argument. The **_fpclassf** function is only available when compiled for the x64 platform.

Requirements

FUNCTION	REQUIRED HEADER
_fpclass, **_fpclassf**	<float.h>

For more compatibility and conformance information, see Compatibility.

See also

Floating-Point Support
isnan, _isnan, _isnanf
fpclassify

fpclassify

11/9/2018 • 2 minutes to read • Edit Online

Returns the floating-point classification of the argument.

Syntax

```
int fpclassify(
   /* floating-point */ x
);

int fpclassify(
   float x
); // C++ only

int fpclassify(
   double x
); // C++ only

int fpclassify(
   long double x
); // C++ only
```

Parameters

x
The floating-point value to test.

Return Value

fpclassify returns an integer value that indicates the floating-point class of the argument *x*. This table shows the possible values returned by **fpclassify**, defined in <math.h>.

VALUE	DESCRIPTION
FP_NAN	A quiet, signaling, or indeterminate NaN
FP_INFINITE	A positive or negative infinity
FP_NORMAL	A positive or negative normalized non-zero value
FP_SUBNORMAL	A positive or negative denormalized value
FP_ZERO	A positive or negative zero value

Remarks

In C, **fpclassify** is a macro; in C++, **fpclassify** is a function overloaded using argument types of **float**, **double**, or **long double**. In either case, the value returned depends on the effective type of the argument expression, and not on any intermediate representation. For example, a normal **double** or **long double** value can become an infinity, denormal, or zero value when converted to a **float**.

Requirements

FUNCTION/MACRO	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
fpclassify	<math.h>	<math.h> or <cmath>

The **fpclassify** macro and **fpclassify** functions conform to the ISO C99 and C++11 specifications. For more compatibility information, see Compatibility.

See also

Floating-Point Support
isnan, _isnan, _isnanf

_fpieee_flt

10/31/2018 • 2 minutes to read • Edit Online

Invokes a user-defined trap handler for IEEE floating-point exceptions.

Syntax

```
int _fpieee_flt(
   unsigned long excCode,
   struct _EXCEPTION_POINTERS *excInfo,
   int handler(_FPIEEE_RECORD *)
);
```

Parameters

excCode
Exception code.

excInfo
Pointer to the Windows NT exception information structure.

handler
Pointer to the user's IEEE trap-handler routine.

Return Value

The return value of **_fpieee_flt** is the value returned by *handler*. As such, the IEEE filter routine might be used in the except clause of a structured exception-handling (SEH) mechanism.

Remarks

The **_fpieee_flt** function invokes a user-defined trap handler for IEEE floating-point exceptions and provides it with all relevant information. This routine serves as an exception filter in the SEH mechanism, which invokes your own IEEE exception handler when necessary.

The **_FPIEEE_RECORD** structure, defined in Fpieee.h, contains information pertaining to an IEEE floating-point exception. This structure is passed to the user-defined trap handler by **_fpieee_flt**.

_FPIEEE_RECORD FIELD	DESCRIPTION
RoundingMode **Precision**	These **unsigned int** fields contain information about the floating-point environment at the time the exception occurred.
Operation	This **unsigned int** field indicates the type of operation that caused the trap. If the type is a comparison (**_FpCodeCompare**), you can supply one of the special **_FPIEEE_COMPARE_RESULT** values (as defined in Fpieee.h) in the **Result.Value** field. The conversion type (**_FpCodeConvert**) indicates that the trap occurred during a floating-point conversion operation. You can look at the **Operand1** and **Result** types to determine the type of conversion being attempted.

_FPIEEE_RECORD FIELD	DESCRIPTION
Operand1 **Operand2** **Result**	These **_FPIEEE_VALUE** structures indicate the types and values of the proposed result and operands. Each structure contains these fields: **OperandValid** - Flag indicating whether the responding value is valid. **Format** - Data type of the corresponding value. The format type might be returned even if the corresponding value is not valid. **Value** - Result or operand data value.
Cause **Enable** **Status**	**_FPIEEE_EXCEPTION_FLAGS** contains one bit field per type of floating point exception. There is a correspondence between these fields and the arguments used to mask the exceptions supplied to _controlfp. The exact meaning of each bit depends on context: **Cause** - Each set bit indicates the particular exception that was raised. **Enable** - Each set bit indicates that the particular exception is currently unmasked. **Status** - Each set bit indicates that the particular exception is currently pending. This includes exceptions that have not been raised because they were masked by **_controlfp**.

Pending exceptions that are disabled are raised when you enable them. This can result in undefined behavior when using **_fpieee_flt** as an exception filter. Always call _clearfp before enabling floating point exceptions.

Requirements

FUNCTION	REQUIRED HEADER
_fpieee_flt	<fpieee.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fpieee.c
// This program demonstrates the implementation of
// a user-defined floating-point exception handler using the
// _fpieee_flt function.

#include <fpieee.h>
#include <excpt.h>
#include <float.h>
#include <stddef.h>

int fpieee_handler( _FPIEEE_RECORD * );

int fpieee_handler( _FPIEEE_RECORD *pieee )
{
   // user-defined ieee trap handler routine:
   // there is one handler for all
   // IEEE exceptions

   // Assume the user wants all invalid
   // operations to return 0.

   if ((pieee->Cause.InvalidOperation) &&
       (pieee->Result.Format == _FpFormatFp32))
   {
        pieee->Result.Value.Fp32Value = 0.0F;

        return EXCEPTION_CONTINUE_EXECUTION;
   }
   else
      return EXCEPTION_EXECUTE_HANDLER;
}

#define _EXC_MASK    \
    _EM_UNDERFLOW  + \
    _EM_OVERFLOW   + \
    _EM_ZERODIVIDE + \
    _EM_INEXACT

int main( void )
{
   // ...

   __try {
      // unmask invalid operation exception
      _controlfp_s(NULL, _EXC_MASK, _MCW_EM);

      // code that may generate
      // fp exceptions goes here
   }
   __except ( _fpieee_flt( GetExceptionCode(),
                GetExceptionInformation(),
                fpieee_handler ) ){

      // code that gets control

      // if fpieee_handler returns
      // EXCEPTION_EXECUTE_HANDLER goes here

   }

   // ...
}
```

See also

Floating-Point Support
_control87, _controlfp, __control87_2
_controlfp_s

_fpreset

10/31/2018 • 2 minutes to read • Edit Online

Resets the floating-point package.

Syntax

```
void _fpreset( void );
```

Remarks

The **_fpreset** function reinitializes the floating-point math package. **_fpreset** is usually used with **signal**, **system**, or the **_exec** or **_spawn** functions. If a program traps floating-point error signals (**SIGFPE**) with **signal**, it can safely recover from floating-point errors by invoking **_fpreset** and using **longjmp**.

This function is deprecated when compiling with /clr (Common Language Runtime Compilation) because the common language runtime only supports the default floating-point precision.

Requirements

FUNCTION	REQUIRED HEADER
_fpreset	<float.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fpreset.c
// This program uses signal to set up a
// routine for handling floating-point errors.

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

jmp_buf mark;              // Address for long jump to jump to
int     fperr;             // Global error number

void __cdecl fphandler( int sig, int num );   // Prototypes
void fpcheck( void );

int main( void )
{
   double n1 = 5.0;
   double n2 = 0.0;
   double r;
   int jmpret;

   // Unmask all floating-point exceptions.
```

```
    _control87( 0, _MCW_EM );
   // Set up floating-point error handler. The compiler
   // will generate a warning because it expects
   // signal-handling functions to take only one argument.
   if( signal( SIGFPE, (void (__cdecl *)(int)) fphandler ) == SIG_ERR )
    {
       fprintf( stderr, "Couldn't set SIGFPE\n" );
       abort();
    }

   // Save stack environment for return in case of error. First
   // time through, jmpret is 0, so true conditional is executed.
   // If an error occurs, jmpret will be set to -1 and false
   // conditional will be executed.
   jmpret = setjmp( mark );
   if( jmpret == 0 )
   {
      printf( "Dividing %4.3g by %4.3g...\n", n1, n2 );
      r = n1 / n2;
      // This won't be reached if error occurs.
      printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );

      r = n1 * n2;
      // This won't be reached if error occurs.
      printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
   }
   else
      fpcheck();
}
// fphandler handles SIGFPE (floating-point error) interrupt. Note
// that this prototype accepts two arguments and that the
// prototype for signal in the run-time library expects a signal
// handler to have only one argument.
//
// The second argument in this signal handler allows processing of
// _FPE_INVALID, _FPE_OVERFLOW, _FPE_UNDERFLOW, and
// _FPE_ZERODIVIDE, all of which are Microsoft-specific symbols
// that augment the information provided by SIGFPE. The compiler
// will generate a warning, which is harmless and expected.

void fphandler( int sig, int num )
{
   // Set global for outside check since we don't want
   // to do I/O in the handler.
   fperr = num;

   // Initialize floating-point package. */
   _fpreset();

   // Restore calling environment and jump back to setjmp. Return
   // -1 so that setjmp will return false for conditional test.
   longjmp( mark, -1 );
}

void fpcheck( void )
{
   char fpstr[30];
   switch( fperr )
   {
   case _FPE_INVALID:
       strcpy_s( fpstr, sizeof(fpstr), "Invalid number" );
       break;
   case _FPE_OVERFLOW:
       strcpy_s( fpstr, sizeof(fpstr), "Overflow" );

       break;
   case _FPE_UNDERFLOW:
       strcpy_s( fpstr, sizeof(fpstr), "Underflow" );
       break;
```

```
    case _FPE_ZERODIVIDE:
        strcpy_s( fpstr, sizeof(fpstr), "Divide by zero" );
        break;
    default:
        strcpy_s( fpstr, sizeof(fpstr), "Other floating point error" );
        break;
    }
    printf( "Error %d: %s\n", fperr, fpstr );
}
```

```
Dividing    5 by    0...
Error 131: Divide by zero
```

See also

Floating-Point Support
_exec, _wexec Functions
signal
_spawn, _wspawn Functions
system, _wsystem

fprintf, _fprintf_l, fwprintf, _fwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Print formatted data to a stream. More secure versions of these functions are available; see fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l.

Syntax

```
int fprintf(
   FILE *stream,
   const char *format [,
   argument ]...
);
int _fprintf_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
   argument ]...
);
int fwprintf(
   FILE *stream,
   const wchar_t *format [,
   argument ]...
);
int _fwprintf_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
   argument ]...
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

fprintf returns the number of bytes written. **fwprintf** returns the number of wide characters written. Each of these functions returns a negative value instead when an output error occurs. If *stream* or *format* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**. The format string is not checked for valid formatting characters as it is when using **fprintf_s** or **fwprintf_s**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

fprintf formats and prints a series of characters and values to the output *stream*. Each function *argument* (if any) is converted and output according to the corresponding format specification in *format*. For **fprintf**, the *format* argument has the same syntax and use that it has in **printf**.

fwprintf is a wide-character version of **fprintf**; in **fwprintf**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fprintf** does not currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ftprintf	**fprintf**	**fprintf**	**fwprintf**
_ftprintf_l	**_fprintf_l**	**_fprintf_l**	**_fwprintf_l**

For more information, see Format Specifications.

Requirements

FUNCTION	REQUIRED HEADER
fprintf, **_fprintf_l**	<stdio.h>
fwprintf, **_fwprintf_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fprintf.c
/* This program uses fprintf to format various
* data and print it to the file named FPRINTF.OUT. It
* then displays FPRINTF.OUT on the screen using the system
* function to invoke the operating-system TYPE command.
*/

#include <stdio.h>
#include <process.h>

FILE *stream;

int main( void )
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';

   fopen_s( &stream, "fprintf.out", "w" );
   fprintf( stream, "%s%c", s, c );
   fprintf( stream, "%d\n", i );
   fprintf( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf.out" );
}
```

```
this is a string
10
1.500000
```

See also

Stream I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
Format Specification Syntax: printf and wprintf Functions

_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Prints formatted data to a stream.

Syntax

```
int _fprintf_p(
   FILE *stream,
   const char *format [,
   argument ]...
);
int _fprintf_p_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
   argument ]...
);
int _fwprintf_p(
   FILE *stream,
   const wchar_t *format [,
   argument ]...
);
int _fwprintf_p_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
   argument ]...
);
```

Parameters

stream
Pointer to the **FILE** structure.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

_fprintf_p and **_fwprintf_p** return the number of characters written or return a negative value when an output error occurs.

Remarks

_fprintf_p formats and prints a series of characters and values to the output *stream*. Each function *argument* (if any) is converted and output according to the corresponding format specification in *format*. For **_fprintf_p**, the *format* argument has the same syntax and use that it has in **_printf_p**. These functions support positional parameters, meaning that the order of the parameters used by the format string can be changed. For more

information about positional parameters, see printf_p Positional Parameters.

_fwprintf_p is a wide-character version of **_fprintf_p**; in **_fwprintf_p**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **_fprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Like the non-secure versions (see fprintf, _fprintf_l, fwprintf, _fwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if either *stream* or *format* is a null pointer or if there are any unknown or badly formed formatting specifiers. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ftprintf_p	**_fprintf_p**	**_fprintf_p**	**_fwprintf_p**
_ftprintf_p_l	**_fprintf_p_l**	**_fprintf_p_l**	**_fwprintf_p_l**

For more information, see Format Specifications.

Requirements

FUNCTION	REQUIRED HEADER
_fprintf_p, **_fprintf_p_l**	<stdio.h>
_fwprintf_p, **_fwprintf_p_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fprintf_p.c
// This program uses _fprintf_p to format various
// data and print it to the file named FPRINTF_P.OUT. It
// then displays FPRINTF_P.OUT on the screen using the system
// function to invoke the operating-system TYPE command.
//

#include <stdio.h>
#include <process.h>

int main( void )
{
    FILE    *stream = NULL;
    int     i = 10;
    double  fp = 1.5;
    char    s[] = "this is a string";
    char    c = '\n';

    // Open the file
    if ( fopen_s( &stream, "fprintf_p.out", "w" ) == 0)
    {
        // Format and print data
        _fprintf_p( stream, "%2$s%1$c", c, s );
        _fprintf_p( stream, "%d\n", i );
        _fprintf_p( stream, "%f\n", fp );

        // Close the file
        fclose( stream );
    }

    // Verify our data
    system( "type fprintf_p.out" );
}
```

```
this is a string
10
1.500000
```

See also

Stream I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
printf_p Positional Parameters
_cprintf_p, _cprintf_p_l, _cwprintf_p, _cwprintf_p_l
_cprintf_s, _cprintf_s_l, _cwprintf_s, _cwprintf_s_l
printf_p Positional Parameters
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Print formatted data to a stream. These are versions of fprintf, _fprintf_l, fwprintf, _fwprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int fprintf_s(
   FILE *stream,
   const char *format [,
   argument_list ]
);
int _fprintf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
   argument_list ]
);
int fwprintf_s(
   FILE *stream,
   const wchar_t *format [,
   argument_list ]
);
int _fwprintf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
   argument_list ]
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

argument_list
Optional arguments to the format string.

locale
The locale to use.

Return Value

fprintf_s returns the number of bytes written. **fwprintf_s** returns the number of wide characters written. Each of these functions returns a negative value instead when an output error occurs.

Remarks

fprintf_s formats and prints a series of characters and values to the output *stream*. Each argument in *argument_list* (if any) is converted and output according to the corresponding format specification in *format*. The *format* argument uses the format specification syntax for printf and wprintf functions.

fwprintf_s is a wide-character version of **fprintf_s**; in **fwprintf_s**, *format* is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Like the non-secure versions (see fprintf, _fprintf_l, fwprintf, _fwprintf_l), these functions validate their parameters and invoke the invalid parameter handler, as described in Parameter Validation, if either *stream* or *format* is a null pointer. The format string itself is also validated. If there are any unknown or badly formed formatting specifiers, these functions generate the invalid parameter exception. In all cases, If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**. See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ftprintf_s	**fprintf_s**	**fprintf_s**	**fwprintf_s**
_ftprintf_s_l	**_fprintf_s_l**	**_fprintf_s_l**	**_fwprintf_s_l**

For more information, see Format Specifications.

Requirements

FUNCTION	REQUIRED HEADER
fprintf_s, **_fprintf_s_l**	<stdio.h>
fwprintf_s, **_fwprintf_s_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fprintf_s.c
// This program uses fprintf_s to format various
// data and print it to the file named FPRINTF_S.OUT. It
// then displays FPRINTF_S.OUT on the screen using the system
// function to invoke the operating-system TYPE command.

#include <stdio.h>
#include <process.h>

FILE *stream;

int main( void )
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';

   fopen_s( &stream, "fprintf_s.out", "w" );
   fprintf_s( stream, "%s%c", s, c );
   fprintf_s( stream, "%d\n", i );
   fprintf_s( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf_s.out" );
}
```

```
this is a string
10
1.500000
```

See also

Stream I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

fputc, fputwc

10/31/2018 • 2 minutes to read • Edit Online

Writes a character to a stream.

Syntax

```
int fputc(
   int c,
   FILE *stream
);
wint_t fputwc(
   wchar_t c,
   FILE *stream
);
```

Parameters

c
Character to be written.

stream
Pointer to **FILE** structure.

Return Value

Each of these functions returns the character written. For **fputc**, a return value of **EOF** indicates an error. For **fputwc**, a return value of **WEOF** indicates an error. If *stream* is **NULL**, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each of these functions writes the single character *c* to a file at the position indicated by the associated file position indicator (if defined) and advances the indicator as appropriate. In the case of **fputc** and **fputwc**, the file is associated with *stream*. If the file cannot support positioning requests or was opened in append mode, the character is appended to the end of the stream.

The two functions behave identically if the stream is opened in ANSI mode. **fputc** does not currently support output into a UNICODE stream.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see_fputc_nolock, _fputwc_nolock.

Routine-specific remarks follow.

ROUTINE	REMARKS
fputc	Equivalent to **putc**, but implemented only as a function, rather than as a function and a macro.

ROUTINE	REMARKS
fputwc	Wide-character version of **fputc**. Writes *c* as a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fputtc	**fputc**	**fputc**	**fputwc**

Requirements

FUNCTION	REQUIRED HEADER
fputc	<stdio.h>
fputwc	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console—**stdin**, **stdout**, and **stderr**—must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_fputc.c
// This program uses fputc
// to send a character array to stdout.

#include <stdio.h>

int main( void )
{
   char strptr1[] = "This is a test of fputc!!\n";
   char *p;

   // Print line to stream using fputc.
   p = strptr1;
   while( (*p != '\0') && fputc( *(p++), stdout ) != EOF ) ;

}
```

```
This is a test of fputc!!
```

See also

Stream I/O
fgetc, fgetwc
putc, putwc

_fputc_nolock, _fputwc_nolock

10/31/2018 • 2 minutes to read • Edit Online

Writes a character to a stream without locking the thread.

Syntax

```
int _fputc_nolock(
   int c,
   FILE *stream
);
wint_t _fputwc_nolock(
   wchar_t c,
   FILE *stream
);
```

Parameters

c
Character to be written.

stream
Pointer to the **FILE** structure.

Return Value

Each of these functions returns the character written. For error information, see fputc, fputwc.

Remarks

_fputc_nolock and **_fputwc_nolock** are identical to **fputc** and **fputwc**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

The two functions behave identically if the stream is opened in ANSI mode. **_fputc_nolock** does not currently support output into a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fputtc_nolock	**_fputc_nolock**	**_fputc_nolock**	**_fputwc_nolock**

Requirements

FUNCTION	REQUIRED HEADER
_fputc_nolock	<stdio.h>
_fputwc_nolock	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console—**stdin**, **stdout**, and **stderr**—must be redirected before C run-time functions can use them in UWP apps. For more compatibility information, see Compatibility.

Example

```
// crt_fputc_nolock.c
// This program uses _fputc_nolock
// to send a character array to stdout.

#include <stdio.h>

int main( void )
{
   char strptr1[] = "This is a test of _fputc_nolock!!\n";
   char *p;

   // Print line to stream using fputc.
   p = strptr1;
   while( (*p != '\0') && _fputc_nolock( *(p++), stdout ) != EOF ) ;

}
```

```
This is a test of _fputc_nolock!!
```

See also

Stream I/O
fgetc, fgetwc
putc, putwc

fputchar

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _fputchar instead.

_fputchar, _fputwchar

10/31/2018 • 2 minutes to read • Edit Online

Writes a character to **stdout**.

Syntax

```
int _fputchar(
   int c
);
wint_t _fputwchar(
   wchar_t c
);
```

Parameters

c
Character to be written.

Return Value

Each of these functions returns the character written. For **_fputchar**, a return value of **EOF** indicates an error. For **_fputwchar**, a return value of **WEOF** indicates an error. If c is **NULL**, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** (or **WEOF**) and set **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Both of these functions writes the single character *c* to **stdout** and advances the indicator as appropriate. **_fputchar** is equivalent to `fputc( stdout )`. It is also equivalent to **putchar**, but implemented only as a function, rather than as a function and a macro. Unlike **fputc** and **putchar**, these functions are not compatible with the ANSI standard.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fputtchar	**_fputchar**	**_fputchar**	**_fputwchar**

Requirements

FUNCTION	REQUIRED HEADER
_fputchar	<stdio.h>
_fputwchar	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are

associated with the console—**stdin**, **stdout**, and **stderr**—must be redirected before C run-time functions can use them in UWP apps. For more compatibility information, see Compatibility.

Example

```
// crt_fputchar.c
// This program uses _fputchar
// to send a character array to stdout.

#include <stdio.h>

int main( void )
{
    char strptr[] = "This is a test of _fputchar!!\n";
    char *p = NULL;

    // Print line to stream using _fputchar.
    p = strptr;
    while( (*p != '\0') && _fputchar( *(p++) ) != EOF )
      ;
}
```

```
This is a test of _fputchar!!
```

See also

Stream I/O
fgetc, fgetwc
putc, putwc

fputs, fputws

10/31/2018 • 2 minutes to read • Edit Online

Writes a string to a stream.

Syntax

```
int fputs(
   const char *str,
   FILE *stream
);
int fputws(
   const wchar_t *str,
   FILE *stream
);
```

Parameters

str
Output string.

stream
Pointer to **FILE** structure.

Return Value

Each of these functions returns a nonnegative value if it is successful. On an error, **fputs** and **fputws** return **EOF**. If *str* or *stream* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and then **fputs** returns **EOF**, and **fputws** returns **WEOF**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each of these functions copies *str* to the output *stream* at the current position. **fputws** copies the wide-character argument *str* to *stream* as a multibyte-character string or a wide-character string according to whether *stream* is opened in text mode or binary mode, respectively. Neither function copies the terminating null character.

The two functions behave identically if the stream is opened in ANSI mode. **fputs** does not currently support output into a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_fputts	**fputs**	**fputs**	**fputws**

Requirements

FUNCTION	REQUIRED HEADER
fputs	<stdio.h>
fputws	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console—**stdin**, **stdout**, and **stderr**—must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_fputs.c
// This program uses fputs to write
// a single line to the stdout stream.

#include <stdio.h>

int main( void )
{
   fputs( "Hello world from fputs.\n", stdout );
}
```

```
Hello world from fputs.
```

See also

Stream I/O
fgets, fgetws
gets, _getws
puts, _putws

fread

2/7/2019 • 2 minutes to read • Edit Online

Reads data from a stream.

Syntax

```
size_t fread(
   void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer
Storage location for data.

size
Item size in bytes.

count
Maximum number of items to be read.

stream
Pointer to **FILE** structure.

Return Value

fread returns the number of full items actually read, which may be less than *count* if an error occurs or if the end of the file is encountered before reaching *count*. Use the **feof** or **ferror** function to distinguish a read error from an end-of-file condition. If *size* or *count* is 0, **fread** returns 0 and the buffer contents are unchanged. If *stream* or *buffer* is a null pointer, **fread** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these error codes.

Remarks

The **fread** function reads up to *count* items of *size* bytes from the input *stream* and stores them in *buffer*. The file pointer associated with *stream* (if there is one) is increased by the number of bytes actually read. If the given stream is opened in text mode, Windows-style newlines are converted into Unix-style newlines. That is, carriage return-linefeed (CRLF) pairs are replaced by single linefeed (LF) characters. The replacement has no effect on the file pointer or the return value. The file-pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

When used on a text mode stream, if the amount of data requested (that is, *size* * *count*) is greater than or equal to the internal **FILE** * buffer size (by default this is 4096 bytes, configurable by using setvbuf), stream data is copied directly into the user-provided buffer, and newline conversion is done in that buffer. Since the converted data may be shorter than the stream data copied into the buffer, data past *buffer*[*return_value* * *size*] (where *return_value* is the return value from **fread**) may contain unconverted data from the file. For this reason, we

recommend you null-terminate character data at *buffer*[*return_value* * *size*] if the intent of the buffer is to act as a C-style string. See fopen for details on the effects of text mode and binary mode.

This function locks out other threads. If you need a non-locking version, use **_fread_nolock**.

Requirements

FUNCTION	REQUIRED HEADER
fread	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fread.c
// This program opens a file named FREAD.OUT and
// writes 25 characters to the file. It then tries to open
// FREAD.OUT and read in 25 characters. If the attempt succeeds,
// the program displays the number of actual items read.

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;

   // Open file in text mode:
   if( fopen_s( &stream, "fread.out", "w+t" ) == 0 )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = (char)('z' - i);
      // Write 25 characters to stream
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );

   }
   else
      printf( "Problem opening the file\n" );

   if( fopen_s( &stream, "fread.out", "r+t" ) == 0 )
   {
      // Attempt to read in 25 characters
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else
      printf( "File could not be opened\n" );
}
```

```
Wrote 25 items
Number of items read = 25
Contents of buffer = zyxwvutsrqponmlkjihgfedcb
```

See also

Stream I/O
Text and Binary File I/O
fopen
fwrite
_read

fread_s

10/31/2018 • 2 minutes to read • Edit Online

Reads data from a stream. This version of fread has security enhancements, as described in Security Features in the CRT.

Syntax

```
size_t fread_s(
   void *buffer,
   size_t bufferSize,
   size_t elementSize,
   size_t count,
   FILE *stream
);
```

Parameters

buffer
Storage location for data.

bufferSize
Size of the destination buffer in bytes.

elementSize
Size of the item to read in bytes.

count
Maximum number of items to be read.

stream
Pointer to **FILE** structure.

Return Value

fread_s returns the number of (whole) items that were read into the buffer, which may be less than *count* if a read error or the end of the file is encountered before *count* is reached. Use the **feof** or **ferror** function to distinguish an error from an end-of-file condition. If *size* or *count* is 0, **fread_s** returns 0 and the buffer contents are unchanged. If *stream* or *buffer* is a null pointer, **fread_s** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

For more information about error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **fread_s** function reads up to *count* items of *elementSize* bytes from the input *stream* and stores them in *buffer*. The file pointer that is associated with *stream* (if there is one) is increased by the number of bytes actually read. If the given stream is opened in text mode, carriage return-linefeed pairs are replaced with single linefeed characters. The replacement has no effect on the file pointer or the return value. The file-pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

This function locks out other threads. If you require a non-locking version, use **_fread_nolock**.

Requirements

FUNCTION	REQUIRED HEADER
fread_s	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fread_s.c
// Command line: cl /EHsc /nologo /W4 crt_fread_s.c
//
// This program opens a file that's named FREAD.OUT and
// writes characters to the file. It then tries to open
// FREAD.OUT and read in characters by using fread_s. If the attempt succeeds,
// the program displays the number of actual items read.

#include <stdio.h>

#define BUFFERSIZE 30
#define DATASIZE 22
#define ELEMENTCOUNT 2
#define ELEMENTSIZE (DATASIZE/ELEMENTCOUNT)
#define FILENAME "FREAD.OUT"

int main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;

   for ( i = 0; i < DATASIZE; i++ )
      list[i] = (char)('z' - i);
   list[DATASIZE] = '\0'; // terminal null so we can print it

   // Open file in text mode:
   if( fopen_s( &stream, FILENAME, "w+t" ) == 0 )
   {
      // Write DATASIZE characters to stream
      printf( "Contents of buffer before write/read:\n\t%s\n\n", list );
      numwritten = fwrite( list, sizeof( char ), DATASIZE, stream );
      printf( "Wrote %d items\n\n", numwritten );
      fclose( stream );
   } else {
      printf( "Problem opening the file\n" );
      return -1;
   }

   if( fopen_s( &stream, FILENAME, "r+t" ) == 0 )   {
      // Attempt to read in characters in 2 blocks of 11
      numread = fread_s( list, BUFFERSIZE, ELEMENTSIZE, ELEMENTCOUNT, stream );
      printf( "Number of %d-byte elements read = %d\n\n", ELEMENTSIZE, numread );
      printf( "Contents of buffer after write/read:\n\t%s\n", list );
      fclose( stream );
   } else {
      printf( "File could not be opened\n" );
      return -1;
   }
}
```

```
Contents of buffer before write/read:
        zyxwvutsrqponmlkjihgfe

Wrote 22 items

Number of 11-byte elements read = 2

Contents of buffer after write/read:
        zyxwvutsrqponmlkjihgfe
```

See also

Stream I/O
fwrite
_read

_fread_nolock

10/31/2018 • 2 minutes to read • Edit Online

Reads data from a stream, without locking other threads.

Syntax

```
size_t _fread_nolock(
   void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer
Storage location for data.

size
Item size in bytes.

count
Maximum number of items to be read.

stream
Pointer to the **FILE** structure.

Return Value

See fread.

Remarks

This function is a non-locking version of **fread**. It is identical to **fread** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fread_nolock	<stdio.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
fwrite
_read

_fread_nolock_s

10/31/2018 • 2 minutes to read • Edit Online

Reads data from a stream, without locking other threads. This version of fread_nolock has security enhancements, as described in Security Features in the CRT.

Syntax

```
size_t _fread_nolock_s(
   void *buffer,
   size_t bufferSize,
   size_t elementSize,
   size_t elementCount,
   FILE *stream
);
```

Parameters

buffer
Storage location for data.

bufferSize
Size of the destination buffer in bytes.

elementSize
Size of the item to read in bytes.

elementCount
Maximum number of items to be read.

stream
Pointer to **FILE** structure.

Return Value

See fread_s.

Remarks

This function is a non-locking version of **fread_s**. It is identical to **fread_s** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fread_nolock_s	C: <stdio.h>; C++: <cstdio> or <stdio.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
fwrite
_read

free

10/31/2018 • 2 minutes to read • Edit Online

Deallocates or frees a memory block.

Syntax

```
void free(
   void *memblock
);
```

Parameters

memblock
Previously allocated memory block to be freed.

Remarks

The **free** function deallocates a memory block (*memblock*) that was previously allocated by a call to **calloc**, **malloc**, or **realloc**. The number of freed bytes is equivalent to the number of bytes requested when the block was allocated (or reallocated, in the case of **realloc**). If *memblock* is **NULL**, the pointer is ignored and **free** immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by **calloc**, **malloc**, or **realloc**) may affect subsequent allocation requests and cause errors.

If an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

After a memory block has been freed, _heapmin minimizes the amount of free memory on the heap by coalescing the unused regions and releasing them back to the operating system. Freed memory that is not released to the operating system is restored to the free pool and is available for allocation again.

When the application is linked with a debug version of the C run-time libraries, **free** resolves to _free_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

free is marked `__declspec(noalias)`, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

To free memory allocated with _malloca, use _freea.

Requirements

FUNCTION	REQUIRED HEADER
free	<stdlib.h> and <malloc.h>

For additional compatibility information, see Compatibility.

Example

See the example for malloc.

See also

Memory Allocation
_alloca
calloc
malloc
realloc
_free_dbg
_heapmin
_freea

_free_dbg

10/31/2018 • 2 minutes to read • Edit Online

Frees a block of memory in the heap (debug version only).

Syntax

```
void _free_dbg(
   void *userData,
   int blockType
);
```

Parameters

userData
Pointer to the allocated memory block to be freed.

blockType
Type of allocated memory block to be freed: **_CLIENT_BLOCK**, **_NORMAL_BLOCK**, or **_IGNORE_BLOCK**.

Remarks

The **_free_dbg** function is a debug version of the free function. When _DEBUG is not defined, each call to **_free_dbg** is reduced to a call to **free**. Both **free** and **_free_dbg** free a memory block in the base heap, but **_free_dbg** accommodates two debugging features: the ability to keep freed blocks in the heap's linked list to simulate low memory conditions and a block type parameter to free specific allocation types.

_free_dbg performs a validity check on all specified files and block locations before performing the free operation. The application is not expected to provide this information. When a memory block is freed, the debug heap manager automatically checks the integrity of the buffers on either side of the user portion and issues an error report if overwriting has occurred. If the **_CRTDBG_DELAY_FREE_MEM_DF** bit field of the _crtDbgFlag flag is set, the freed block is filled with the value 0xDD, assigned the **_FREE_BLOCK** block type, and kept in the heap's linked list of memory blocks.

If an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_free_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Example

For a sample of how to use **_free_dbg**, see crt_dbg2.

See also

Debug Routines
_malloc_dbg

_free_locale

10/31/2018 • 2 minutes to read • Edit Online

Frees a locale object.

Syntax

```
void _free_locale(
   _locale_t locale
);
```

Parameters

locale
Locale object to free.

Remarks

The **_free_locale** function is used to free the locale object obtained from a call to **_get_current_locale** or **_create_locale**.

The previous name of this function, **__free_locale** (with two leading underscores) has been deprecated.

Requirements

ROUTINE	REQUIRED HEADER
_free_locale	<locale.h>

For more compatibility information, see Compatibility.

See also

_get_current_locale
_create_locale, _wcreate_locale

_freea

10/31/2018 • 2 minutes to read • Edit Online

Deallocates or frees a memory block.

Syntax

```
void _freea(
   void *memblock
);
```

Parameters

memblock
Previously allocated memory block to be freed.

Return Value

None.

Remarks

The **_freea** function deallocates a memory block (*memblock*) that was previously allocated by a call to _malloca. **_freea** checks to see if the memory was allocated on the heap or the stack. If it was allocated on the stack, **_freea** does nothing. If it was allocated on the heap, the number of freed bytes is equivalent to the number of bytes requested when the block was allocated. If *memblock* is **NULL**, the pointer is ignored and **_freea** immediately returns. Attempting to free an invalid pointer (a pointer to a memory block that was not allocated by **_malloca**) might affect subsequent allocation requests and cause errors.

_freea calls **free** internally if it finds that the memory is allocated on the heap. Whether the memory is on the heap or the stack is determined by a marker placed in memory at the address immediately preceding the allocated memory.

If an error occurs in freeing the memory, **errno** is set with information from the operating system on the nature of the failure. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

After a memory block has been freed, _heapmin minimizes the amount of free memory on the heap by coalescing the unused regions and releasing them back to the operating system. Freed memory that is not released to the operating system is restored to the free pool and is available for allocation again.

A call to **_freea** must accompany all calls to **_malloca**. It is also an error to call **_freea** twice on the same memory. When the application is linked with a debug version of the C run-time libraries, particularly with _malloc_dbg features enabled by defining **_CRTDBG_MAP_ALLOC**, it is easier to find missing or duplicated calls to **_freea**. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

_freea is marked `__declspec(noalias)`, meaning that the function is guaranteed not to modify global variables. For more information, see noalias.

Requirements

FUNCTION	REQUIRED HEADER
_freea	<stdlib.h> and <malloc.h>

For more compatibility information, see Compatibility.

Example

See the example for _malloca.

See also

Memory Allocation
_malloca
calloc
malloc
_malloc_dbg
realloc
_free_dbg
_heapmin

freopen, _wfreopen

11/8/2018 • 4 minutes to read • Edit Online

Reassigns a file pointer. More secure versions of these functions are available; see freopen_s, _wfreopen_s.

Syntax

```
FILE *freopen(
   const char *path,
   const char *mode,
   FILE *stream
);
FILE *_wfreopen(
   const wchar_t *path,
   const wchar_t *mode,
   FILE *stream
);
```

Parameters

path
Path of new file.

mode
Type of access permitted.

stream
Pointer to **FILE** structure.

Return Value

Each of these functions returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a **NULL** pointer value. If *path*, *mode*, or *stream* is a null pointer, or if *filename* is an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **NULL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

More secure versions of these functions exist, see freopen_s, _wfreopen_s.

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *path*. **_wfreopen** is a wide-character version of **_freopen**; the *path* and *mode* arguments to **_wfreopen** are wide-character strings. **_wfreopen** and **_freopen** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfreopen	**freopen**	**freopen**	**_wfreopen**

freopen is typically used to redirect the pre-opened files **stdin**, **stdout**, and **stderr** to files specified by the

user. The new file associated with *stream* is opened with *mode*, which is a character string specifying the type of access requested for the file, as follows:

MODE	ACCESS
"r"	Opens for reading. If the file does not exist or cannot be found, the **freopen** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending) without removing the end-of-file (EOF) marker before new data is written to the file. Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. The appending operation includes the removal of the EOF marker before new data is written to the file. The EOF marker is not restored after writing is completed. Creates the file if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations take place at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired. In addition to the above values, one of the following characters may be included in the *mode* string to specify the translation mode for new lines.

MODE MODIFIER	TRANSLATION MODE
t	Open in text (translated) mode.
b	Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

In text (translated) mode, carriage return-linefeed (CR-LF) combinations are translated into single linefeed (LF) characters on input; LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or for writing and reading with **"a+"**, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using fseek and ftell to move within a file may cause fseek to behave improperly near the end of the

file. The **t** option is a Microsoft extension that should not be used where ANSI portability is desired.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Requirements

FUNCTION	REQUIRED HEADER
freopen	<stdio.h>
_wfreopen	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_freopen.c
// compile with: /W3
// This program reassigns stderr to the file
// named FREOPEN.OUT and writes a line to that file.
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main( void )
{
   // Reassign "stderr" to "freopen.out":
   stream = freopen( "freopen.out", "w", stderr ); // C4996
   // Note: freopen is deprecated; consider using freopen_s instead

   if( stream == NULL )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stdout, "successfully reassigned\n" ); fflush( stdout );
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

```
successfully reassigned
This will go to the file 'freopen.out'
```

See also

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
_fileno
fopen, _wfopen

_open, _wopen

_setmode

freopen_s, _wfreopen_s

11/8/2018 • 4 minutes to read • Edit Online

Reassigns a file pointer. These versions of freopen, _wfreopen have security enhancements, as described in Security Features in the CRT.

Syntax

```
errno_t freopen(
   FILE** pFile,
   const char *path,
   const char *mode,
   FILE *stream
);
errno_t _wfreopen(
   FILE** pFile,
   const wchar_t *path,
   const wchar_t *mode,
   FILE *stream
);
```

Parameters

pFile
A pointer to the file pointer to be provided by the call.

path
Path of new file.

mode
Type of access permitted.

stream
Pointer to **FILE** structure.

Return Value

Each of these functions returns an error code. If an error occurs, the original file is closed.

Remarks

The **freopen_s** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *path*. **_wfreopen_s** is a wide-character version of **_freopen_s**; the *path* and *mode* arguments to **_wfreopen_s** are wide-character strings. **_wfreopen_s** and **_freopen_s** behave identically otherwise.

If any of *pFile*, *path*, *mode*, or *stream* are **NULL**, or if *path* is an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfreopen_s	**freopen_s**	**freopen_s**	**_wfreopen_s**

freopen_s is typically used to redirect the pre-opened files **stdin**, **stdout**, and **stderr** to files specified by the user. The new file associated with *stream* is opened with *mode*, which is a character string specifying the type of access requested for the file, as follows:

MODE	ACCESS
"r"	Opens for reading. If the file does not exist or cannot be found, the **freopen_s** call fails.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending) without removing the end-of-file (EOF) marker before new data is written to the file. Creates the file if it does not exist.
"r+"	Opens for both reading and writing. The file must exist.
"w+"	Opens an empty file for both reading and writing. If the file exists, its contents are destroyed.
"a+"	Opens for reading and appending. The appending operation includes the removal of the EOF marker before new data is written to the file. The EOF marker is not restored after writing is completed. Creates the file if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations take place at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

The **"a"** mode does not remove the EOF marker before appending to the file. After appending has occurred, the MS-DOS TYPE command only shows data up to the original EOF marker and not any data appended to the file. The **"a+"** mode does remove the EOF marker before appending to the file. After appending, the MS-DOS TYPE command shows all data in the file. The **"a+"** mode is required for appending to a stream file that is terminated with the CTRL+Z EOF marker.

When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired. In addition to the above values, one of the following characters may be included in the *mode* string to specify the translation mode for new lines.

MODE MODIFIER	TRANSLATION MODE
t	Open in text (translated) mode.
b	Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.

In text (translated) mode, carriage return-linefeed (CR-LF) combinations are translated into single linefeed (LF) characters on input; LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or for writing and reading with **"a+"**, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using fseek and ftell to move within a file may cause fseek to behave improperly near the end of the file. The **t** option is a Microsoft extension that should not be used where ANSI portability is desired.

If **t** or **b** is not given in *mode*, the default translation mode is defined by the global variable _fmode. If **t** or **b** is prefixed to the argument, the function fails and returns **NULL**.

For a discussion of text and binary modes, see Text and Binary Mode File I/O.

Requirements

FUNCTION	REQUIRED HEADER
freopen_s	<stdio.h>
_wfreopen_s	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_freopen_s.c
// This program reassigns stderr to the file
// named FREOPEN.OUT and writes a line to that file.

#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main( void )
{
   errno_t err;
   // Reassign "stderr" to "freopen.out":
   err = freopen_s( &stream, "freopen.out", "w", stderr );

   if( err != 0 )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stdout, "successfully reassigned\n" ); fflush( stdout );
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

```
successfully reassigned
This will go to the file 'freopen.out'
```

See also

Stream I/O
freopen, _wfreopen
fclose, _fcloseall
_fdopen, _wfdopen
_fileno
fopen, _wfopen
_open, _wopen
_setmode

frexp, frexpf, frexpl

10/31/2018 • 2 minutes to read • Edit Online

Gets the mantissa and exponent of a floating-point number.

Syntax

```
double frexp(
   double x,
   int *expptr
);
float frexpf(
   float x,
   int * expptr
);
long double frexpl(
   long double x,
   int * expptr
);
float frexp(
   float x,
   int * expptr
);  // C++ only
long double frexp(
   long double x,
   int * expptr
);  // C++ only
```

Parameters

x
Floating-point value.

expptr
Pointer to stored integer exponent.

Return Value

frexp returns the mantissa. If *x* is 0, the function returns 0 for both the mantissa and the exponent. If *expptr* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

Remarks

The **frexp** function breaks down the floating-point value (*x*) into a mantissa (*m*) and an exponent (*n*), such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0, and $x = m * 2^n$. The integer exponent *n* is stored at the location pointed to by *expptr*.

C++ allows overloading, so you can call overloads of **frexp**. In a C program, **frexp** always takes a **double** and an **int** pointer and returns a **double**.

Requirements

FUNCTION	REQUIRED HEADER
frexp, **frexpf**, **frexpl**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_frexp.c
// This program calculates frexp( 16.4, &n )
// then displays y and n.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x, y;
   int n;

   x = 16.4;
   y = frexp( x, &n );
   printf( "frexp( %f, &n ) = %f, n = %d\n", x, y, n );
}
```

```
frexp( 16.400000, &n ) = 0.512500, n = 5
```

See also

Floating-Point Support
ldexp
modf, modff, modfl

fscanf, _fscanf_l, fwscanf, _fwscanf_l

10/31/2018 • 2 minutes to read • Edit Online

Read formatted data from a stream. More secure versions of these functions are available; see fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l.

Syntax

```
int fscanf(
   FILE *stream,
   const char *format [,
   argument ]...
);
int _fscanf_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
   argument ]...
);
int fwscanf(
   FILE *stream,
   const wchar_t *format [,
   argument ]...
);
int _fwscanf_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
   argument ]...
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **fscanf** and **fwscanf**.

These functions validate their parameters. If *stream* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **fscanf** function reads data from the current position of *stream* into the locations given by *argument* (if any). Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf**; see scanf for a description of *format*.

fwscanf is a wide-character version of **fscanf**; the format argument to **fwscanf** is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fscanf** doesn't currently support input from a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ftscanf	**fscanf**	**fscanf**	**fwscanf**
_ftscanf_l	**_fscanf_l**	**_fscanf_l**	**_fwscanf_l**

For more information, see Format Specification Fields - scanf functions and wscanf Functions.

Requirements

FUNCTION	REQUIRED HEADER
fscanf, **_fscanf_l**	<stdio.h>
fwscanf, **_fwscanf_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fscanf.c
// compile with: /W3
// This program writes formatted
// data to a file. It then uses fscanf to
// read the various data back from the file.

#include <stdio.h>

FILE *stream;

int main( void )
{
   long l;
   float fp;
   char s[81];
   char c;

   if( fopen_s( &stream, "fscanf.out", "w+" ) != 0 )
      printf( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf( stream, "%s %ld %f%c", "a-string",
               65000, 3.14159, 'x' );
      // Security caution!
      // Beware loading data from a file without confirming its size,
      // as it may lead to a buffer overrun situation.

      // Set pointer to beginning of file:
      fseek( stream, 0L, SEEK_SET );

      // Read data back from file:
      fscanf( stream, "%s", s );   // C4996
      fscanf( stream, "%ld", &l ); // C4996

      fscanf( stream, "%f", &fp ); // C4996
      fscanf( stream, "%c", &c );  // C4996
      // Note: fscanf is deprecated; consider using fscanf_s instead

      // Output data read:
      printf( "%s\n", s );
      printf( "%ld\n", l );
      printf( "%f\n", fp );
      printf( "%c\n", c );

      fclose( stream );
   }
}
```

```
a-string
65000
3.141590
x
```

See also

Stream I/O
_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data from a stream. These versions of fscanf, _fscanf_l, fwscanf, _fwscanf_l have security enhancements, as described in Security Features in the CRT.

Syntax

```
int fscanf_s(
   FILE *stream,
   const char *format [,
   argument ]...
);
int _fscanf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale [,
   argument ]...
);
int fwscanf_s(
   FILE *stream,
   const wchar_t *format [,
   argument ]...
);
int _fwscanf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale [,
   argument ]...
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **fscanf_s** and **fwscanf_s**.

These functions validate their parameters. If *stream* is an invalid file pointer, or *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **fscanf_s** function reads data from the current position of *stream* into the locations that are given by *argument* (if any). Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf_s**; see Format Specification Fields: scanf and wscanf Functions for a description of *format*. **fwscanf_s** is a wide-character version of **fscanf_s**; the format argument to **fwscanf_s** is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **fscanf_s** doesn't currently support input from a UNICODE stream.

The main difference between the more secure functions (that have the **_s** suffix) and the other versions is that the more secure functions require the size in characters of each **c**, **C**, **s**, **S**, and **[** type field to be passed as an argument immediately following the variable. For more information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Width Specification.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions that have the **_l** suffix are identical except that they use the locale parameter that's passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ftscanf_s	**fscanf_s**	**fscanf_s**	**fwscanf_s**
_ftscanf_s_l	**_fscanf_s_l**	**_fscanf_s_l**	**_fwscanf_s_l**

Requirements

FUNCTION	REQUIRED HEADER
fscanf_s, **_fscanf_s_l**	<stdio.h>
fwscanf_s, **_fwscanf_s_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fscanf_s.c
// This program writes formatted
// data to a file. It then uses fscanf to
// read the various data back from the file.

#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main( void )
{
   long l;
   float fp;
   char s[81];
   char c;

   errno_t err = fopen_s( &stream, "fscanf.out", "w+" );
   if( err )
      printf_s( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf_s( stream, "%s %ld %f%c", "a-string",
                65000, 3.14159, 'x' );
      // Set pointer to beginning of file:
      fseek( stream, 0L, SEEK_SET );

      // Read data back from file:
      fscanf_s( stream, "%s", s, _countof(s) );
      fscanf_s( stream, "%ld", &l );

      fscanf_s( stream, "%f", &fp );
      fscanf_s( stream, "%c", &c, 1 );

      // Output data read:
      printf( "%s\n", s );
      printf( "%ld\n", l );
      printf( "%f\n", fp );
      printf( "%c\n", c );

      fclose( stream );
   }
}
```

```
a-string
65000
3.141590
x
```

See also

Stream I/O
_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l
fscanf, _fscanf_l, fwscanf, _fwscanf_l

fseek, _fseeki64

11/8/2018 • 3 minutes to read • Edit Online

Moves the file pointer to a specified location.

Syntax

```
int fseek(
   FILE *stream,
   long offset,
   int origin
);
int _fseeki64(
   FILE *stream,
   __int64 offset,
   int origin
);
```

Parameters

stream
Pointer to **FILE** structure.

offset
Number of bytes from *origin*.

origin
Initial position.

Return Value

If successful, **fseek** and **_fseeki64** returns 0. Otherwise, it returns a nonzero value. On devices incapable of seeking, the return value is undefined. If *stream* is a null pointer, or if *origin* is not one of allowed values described below, **fseek** and **_fseeki64** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

Remarks

The **fseek** and **_fseeki64** functions moves the file pointer (if any) associated with *stream* to a new location that is *offset* bytes from *origin*. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write. The argument *origin* must be one of the following constants, defined in STDIO.H:

ORIGIN VALUE	MEANING
SEEK_CUR	Current position of file pointer.
SEEK_END	End of file.
SEEK_SET	Beginning of file.

You can use **fseek** and **_fseeki64** to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. **fseek** and **_fseeki64** clears the end-of-file indicator and negates the effect of any prior ungetc calls against *stream*.

When a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

For streams opened in text mode, **fseek** and **_fseeki64** have limited use, because carriage return-linefeed translations can cause **fseek** and **_fseeki64** to produce unexpected results. The only **fseek** and **_fseeki64** operations guaranteed to work on streams opened in text mode are:

- Seeking with an offset of 0 relative to any of the origin values.
- Seeking from the beginning of the file with an offset value returned from a call to ftell when using **fseek** or _ftelli64 when using **_fseeki64**.

Also in text mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing, fopen and all related routines check for a CTRL+Z at the end of the file and remove it if possible. This is done because using the combination of **fseek** and ftell or **_fseeki64** and _ftelli64, to move within a file that ends with a CTRL+Z may cause **fseek** or **_fseeki64** to behave improperly near the end of the file.

When the CRT opens a file that begins with a Byte Order Mark (BOM), the file pointer is positioned after the BOM (that is, at the start of the file's actual content). If you have to **fseek** to the beginning of the file, use ftell to get the initial position and **fseek** to it rather than to position 0.

This function locks out other threads during execution and is therefore thread-safe. For a non-locking version, see _fseek_nolock, _fseeki64_nolock.

Requirements

FUNCTION	REQUIRED HEADER
fseek	<stdio.h>
_fseeki64	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_fseek.c
// This program opens the file FSEEK.OUT and
// moves the pointer to the file's beginning.

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char line[81];
   int  result;

   if ( fopen_s( &stream, "fseek.out", "w+" ) != 0 )
   {
      printf( "The file fseek.out was not opened\n" );
      return -1;
   }
   fprintf( stream, "The fseek begins here: "
                    "This is the file 'fseek.out'.\n" );
   result = fseek( stream, 23L, SEEK_SET);
   if( result )
      perror( "Fseek failed" );
   else
   {
      printf( "File pointer is set to middle of first line.\n" );
      fgets( line, 80, stream );
      printf( "%s", line );
    }
   fclose( stream );
}
```

```
File pointer is set to middle of first line.
This is the file 'fseek.out'.
```

See also

Stream I/O
fopen, _wfopen
ftell, _ftelli64
_lseek, _lseeki64
rewind

_fseek_nolock, _fseeki64_nolock

10/31/2018 • 2 minutes to read • Edit Online

Moves the file pointer to a specified location.

Syntax

```
int _fseek_nolock(
   FILE *stream,
   long offset,
   int origin
);
int _fseeki64_nolock(
   FILE *stream,
   __int64 offset,
   int origin
);
```

Parameters

stream
Pointer to the **FILE** structure.

offset
Number of bytes from *origin*.

origin
Initial position.

Return Value

Same as fseek and _fseeki64, respectively.

Remarks

These functions are the non-locking versions of fseek and _fseeki64, respectively. These are identical to fseek and _fseeki64 except that they are not protected from interference by other threads. These functions might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fseek_nolock, **_fseeki64_nolock**	<stdio.h>

For additional compatibility information, see Compatibility.

See also

Stream I/O
ftell, _ftelli64

_lseek, _lseeki64
rewind

fsetpos

10/31/2018 • 2 minutes to read • Edit Online

Sets the stream-position indicator.

Syntax

```
int fsetpos(
   FILE *stream,
   const fpos_t *pos
);
```

Parameters

stream
Pointer to **FILE** structure.

pos
Position-indicator storage.

Return Value

If successful, **fsetpos** returns 0. On failure, the function returns a nonzero value and sets **errno** to one of the following manifest constants (defined in ERRNO.H): **EBADF**, which means the file is not accessible or the object that *stream* points to is not a valid file structure; or **EINVAL**, which means an invalid value for *stream* or *pos* was passed. If an invalid parameter is passed in, these functions invoke the invalid parameter handler, as described in Parameter Validation.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **fsetpos** function sets the file-position indicator for *stream* to the value of *pos*, which is obtained in a prior call to **fgetpos** against *stream*. The function clears the end-of-file indicator and undoes any effects of ungetc on *stream*. After calling **fsetpos**, the next operation on *stream* may be either input or output.

Requirements

FUNCTION	REQUIRED HEADER
fsetpos	<stdio.h>

For additional compatibility information, see Compatibility.

Example

See the example for fgetpos.

See also

Stream I/O

_fsopen, _wfsopen

10/31/2018 • 4 minutes to read • Edit Online

Opens a stream with file sharing.

Syntax

```
FILE *_fsopen(
   const char *filename,
   const char *mode,
   int shflag
);
FILE *_wfsopen(
   const wchar_t *filename,
   const wchar_t *mode,
   int shflag
);
```

Parameters

filename
Name of the file to open.

mode
Type of access permitted.

shflag
Type of sharing allowed.

Return Value

Each of these functions returns a pointer to the stream. A null pointer value indicates an error. If *filename* or *mode* is **NULL** or an empty string, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_fsopen** function opens the file specified by *filename* as a stream and prepares the file for subsequent shared reading or writing, as defined by the mode and *shflag* arguments. **_wfsopen** is a wide-character version of **_fsopen**; the *filename* and *mode* arguments to **_wfsopen** are wide-character strings. **_wfsopen** and **_fsopen** behave identically otherwise.

The character string *mode* specifies the type of access requested for the file, as shown in the following table.

TERM	DEFINITION
"r"	Opens for reading. If the file does not exist or cannot be found, the **_fsopen** call fails.

TERM	DEFINITION
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a"	Opens for writing at the end of the file (appending); creates the file first if it does not exist.
"r+"	Opens for both reading and writing. (The file must exist.)
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Opens for reading and appending; creates the file first if it does not exist.

Use the **"w"** and **"w+"** types with care, as they can destroy existing files.

When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file. The file pointer can be repositioned using fseek or rewind, but it is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten. When the **"r+"**, **"w+"**, or **"a+"** access type is specified, both reading and writing are allowed (the file is said to be open for update). However, when switching between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired. In addition to the above values, one of the following characters can be included in *mode* to specify the translation mode for new lines, and for file management.

TERM	DEFINITION
t	Opens a file in text (translated) mode. In this mode, carriage return-line feed (CR-LF) combinations are translated into single line feeds (LF) on input and LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or reading/writing, **_fsopen** checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using fseek and ftell to move within a file that ends with a CTRL+Z might cause fseek to behave improperly near the end of the file.
b	Opens a file in binary (untranslated) mode; the above translations are suppressed.
S	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
R	Specifies that caching is optimized for, but not restricted to, random access from disk.
T	Specifies a file as temporary. If possible, it is not flushed to disk.
D	Specifies a file as temporary. It is deleted when the last file pointer is closed.

If **t** or **b** is not given in *mode*, the translation mode is defined by the default-mode variable **_fmode**. If **t** or **b** is

prefixed to the argument, the function fails and returns **NULL**. For a discussion of text and binary modes, see Text and Binary Mode File I/O.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in Share.h.

TERM	DEFINITION
_SH_COMPAT	Sets Compatibility mode for 16-bit applications.
_SH_DENYNO	Permits read and write access.
_SH_DENYRD	Denies read access to the file.
_SH_DENYRW	Denies read and write access to the file.
_SH_DENYWR	Denies write access to the file.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfsopen	**_fsopen**	**_fsopen**	**_wfsopen**

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADERS
_fsopen	<stdio.h>	<share.h> For manifest constant for *shflag* parameter.
_wfsopen	<stdio.h> or <wchar.h>	<share.h> For manifest constant for *shflag* parameter.

Example

```
// crt_fsopen.c

#include <stdio.h>
#include <stdlib.h>
#include <share.h>

int main( void )
{
   FILE *stream;

   // Open output file for writing. Using _fsopen allows us to
   // ensure that no one else writes to the file while we are
   // writing to it.
    //
   if( (stream = _fsopen( "outfile", "wt", _SH_DENYWR )) != NULL )
   {
      fprintf( stream, "No one else in the network can write "
                       "to this file until we are done.\n" );
      fclose( stream );
   }
   // Now others can write to the file while we read it.
   system( "type outfile" );
}
```

```
No one else in the network can write to this file until we are done.
```

See also

Stream I/O
fclose, _fcloseall
_fdopen, _wfdopen
ferror
_fileno
fopen, _wfopen
freopen, _wfreopen
_open, _wopen
_setmode
_sopen, _wsopen

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

11/8/2018 • 3 minutes to read • Edit Online

Gets information about an open file.

Syntax

```
int _fstat(
   int fd,
   struct _stat *buffer
);
int _fstat32(
   int fd,
   struct __stat32 *buffer
);
int _fstat64(
   int fd,
   struct __stat64 *buffer
);
int _fstati64(
   int fd,
   struct _stati64 *buffer
);
int _fstat32i64(
   int fd,
   struct _stat32i64 *buffer
);
int _fstat64i32(
   int fd,
   struct _stat64i32 *buffer
);
```

Parameters

fd
File descriptor of the open file.

buffer
Pointer to the structure to store results.

Return Value

Returns 0 if the file-status information is obtained. A return value of -1 indicates an error. If the file descriptor is invalid or *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EBADF**, in the case of an invalid file descriptor, or to **EINVAL**, if *buffer* is **NULL**.

Remarks

The **_fstat** function obtains information about the open file associated with *fd* and stores it in the structure pointed to by *buffer*. The **_stat** structure, defined in SYS\Stat.h, contains the following fields.

FIELD	MEANING
st_atime	Time of the last file access.
st_ctime	Time of the creation of the file.
st_dev	If a device, *fd*; otherwise 0.
st_mode	Bit mask for file-mode information. The **_S_IFCHR** bit is set if *fd* refers to a device. The **_S_IFREG** bit is set if *fd* refers to an ordinary file. The read/write bits are set according to the file's permission mode. **_S_IFCHR** and other constants are defined in SYS\Stat.h.
st_mtime	Time of the last modification of the file.
st_nlink	Always 1 on non-NTFS file systems.
st_rdev	If a device, *fd*; otherwise 0.
st_size	Size of the file in bytes.

If *fd* refers to a device, the **st_atime**, **st_ctime**, **st_mtime**, and **st_size** fields are not meaningful.

Because Stat.h uses the _dev_t type, which is defined in Types.h, you must include Types.h before Stat.h in your code.

_fstat64, which uses the **__stat64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas the other functions only represent dates through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

Variations of these functions support 32-bit or 64-bit time types and 32-bit or 64-bit file lengths. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer.

_fstat is equivalent to **_fstat64i32**, and **struct _stat** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_fstat** uses a 32-bit time, and **struct _stat** contains a 32-bit time. The same is true for **_fstati64**.

Time Type and File Length Type Variations of _stat

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_fstat	Not defined	64-bit	32-bit
_fstat	Defined	32-bit	32-bit
_fstat32	Not affected by the macro definition	32-bit	32-bit
_fstat64	Not affected by the macro definition	64-bit	64-bit
_fstati64	Not defined	64-bit	64-bit

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_fstati64	Defined	32-bit	64-bit
_fstat32i64	Not affected by the macro definition	32-bit	64-bit
_fstat64i32	Not affected by the macro definition	64-bit	32-bit

Requirements

FUNCTION	REQUIRED HEADER
_fstat	<sys/stat.h> and <sys/types.h>
_fstat32	<sys/stat.h> and <sys/types.h>
_fstat64	<sys/stat.h> and <sys/types.h>
_fstati64	<sys/stat.h> and <sys/types.h>
_fstat32i64	<sys/stat.h> and <sys/types.h>
_fstat64i32	<sys/stat.h> and <sys/types.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fstat.c
// This program uses _fstat to report
// the size of a file named F_STAT.OUT.

#include <io.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <share.h>

int main( void )
{
   struct _stat buf;
   int fd, result;
   char buffer[] = "A line to output";
   char timebuf[26];
   errno_t err;

   _sopen_s( &fd,
             "f_stat.out",
             _O_CREAT | _O_WRONLY | _O_TRUNC,
             _SH_DENYNO,
             _S_IREAD | _S_IWRITE );
   if( fd != -1 )
      _write( fd, buffer, strlen( buffer ) );

   // Get data associated with "fd":
   result = _fstat( fd, &buf );

   // Check if statistics are valid:
   if( result != 0 )
   {
      if (errno == EBADF)
        printf( "Bad file descriptor.\n" );
      else if (errno == EINVAL)
        printf( "Invalid argument to _fstat.\n" );
   }
   else
   {
      printf( "File size     : %ld\n", buf.st_size );
      err = ctime_s(timebuf, 26, &buf.st_mtime);
      if (err)
      {
         printf("Invalid argument to ctime_s.");
         exit(1);
      }
      printf( "Time modified : %s", timebuf );
   }
   _close( fd );
}
```

```
File size     : 16
Time modified : Wed May 07 15:25:11 2003
```

See also

File Handling

_access, _waccess

_chmod, _wchmod
_filelength, _filelengthi64
_stat, _wstat Functions

ftell, _ftelli64

10/31/2018 • 2 minutes to read • Edit Online

Gets the current position of a file pointer.

Syntax

```
long ftell(
   FILE *stream
);
__int64 _ftelli64(
   FILE *stream
);
```

Parameters

stream
Target **FILE** structure.

Return Value

ftell and **_ftelli64** return the current file position. The value returned by **ftell** and **_ftelli64** may not reflect the physical byte offset for streams opened in text mode, because text mode causes carriage return-linefeed translation. Use **ftell** with fseek or **_ftelli64** with _fseeki64 to return to file locations correctly. On error,**ftell** and **_ftelli64** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1L and set **errno** to one of two constants, defined in ERRNO.H. The **EBADF** constant means the *stream* argument is not a valid file pointer value or does not refer to an open file. **EINVAL** means an invalid *stream* argument was passed to the function. On devices incapable of seeking (such as terminals and printers), or when *stream* does not refer to an open file, the return value is undefined.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **ftell** and **_ftelli64** functions retrieve the current position of the file pointer (if any) associated with *stream*. The position is expressed as an offset relative to the beginning of the stream.

Note that when a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. For example, if a file is opened for an append and the last operation was a read, the file position is the point where the next read operation would start, not where the next write would start. (When a file is opened for appending, the file position is moved to end of file before any write operation.) If no I/O operation has yet occurred on a file opened for appending, the file position is the beginning of the file.

In text mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing, **fopen** and all related routines check for a CTRL+Z at the end of the file and remove it if possible. This is done because using the combination of **ftell** and fseek or **_ftelli64** and _fseeki64, to move within a file that ends with a CTRL+Z may cause **ftell** or **_ftelli64** to behave improperly near the end of the file.

This function locks the calling thread during execution and is therefore thread-safe. For a non-locking version, see **_ftell_nolock**.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADERS
ftell	<stdio.h>	<errno.h>
_ftelli64	<stdio.h>	<errno.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_ftell.c
// This program opens a file named CRT_FTELL.C
// for reading and tries to read 100 characters. It
// then uses ftell to determine the position of the
// file pointer and displays this position.

#include <stdio.h>

FILE *stream;

int main( void )
{
   long position;
   char list[100];
   if( fopen_s( &stream, "crt_ftell.c", "rb" ) == 0 )
   {
      // Move the pointer by reading data:
      fread( list, sizeof( char ), 100, stream );
      // Get position after read:
      position = ftell( stream );
      printf( "Position after trying to read 100 bytes: %ld\n",
              position );
      fclose( stream );
   }
}
```

```
Position after trying to read 100 bytes: 100
```

See also

Stream I/O
fopen, _wfopen
fgetpos
fseek, _fseeki64
_lseek, _lseeki64

_ftell_nolock, _ftelli64_nolock

10/31/2018 • 2 minutes to read • Edit Online

Gets the current position of a file pointer, without locking the thread.

Syntax

```
long _ftell_nolock(
   FILE *stream
);
__int64 _ftelli64_nolock(
   FILE *stream
);
```

Parameters

stream
Target the **FILE** structure.

Return Value

Same as **ftell** and **_ftelli64**. For more information, see ftell, _ftelli64.

Remarks

These functions are non-locking versions of **ftell** and **_ftelli64**, respectively. They are identical to **ftell** and **_ftelli64** except that they are not protected from interference by other threads. These functions might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
ftell_nolock	<stdio.h>	<errno.h>
_ftelli64_nolock	<stdio.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
fgetpos
fseek, _fseeki64
_lseek, _lseeki64
ftell, _ftelli64

_ftime, _ftime32, _ftime64

10/31/2018 • 2 minutes to read • Edit Online

Get the current time. More secure versions of these functions are available; see _ftime_s, _ftime32_s, _ftime64_s.

Syntax

```
void _ftime( struct _timeb *timeptr );
void _ftime32( struct __timeb32 *timeptr );
void _ftime64( struct __timeb64 *timeptr );
```

Parameters

timeptr
Pointer to a **_timeb**, **__timeb32**, or **__timeb64** structure.

Remarks

The **_ftime** function gets the current local time and stores it in the structure pointed to by *timeptr*. The **_timeb**, **__timeb32**, and **__timeb64** structures are defined in <sys\timeb.h>. They contain four fields, which are listed in the following table.

FIELD	DESCRIPTION
dstflag	Nonzero if daylight savings time is currently in effect for the local time zone. (See _tzset for an explanation of how daylight savings time is determined.)
millitm	Fraction of a second in milliseconds.
time	Time in seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).
timezone	Difference in minutes, moving westward, between UTC and local time. The value of **timezone** is set from the value of the global variable **_timezone** (see **_tzset**).

The **_ftime64** function, which uses the **__timeb64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **_ftime32** only represents dates through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

The **_ftime** function is equivalent to **_ftime64**, and **_timeb** contains a 64-bit time unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_ftime** uses a 32-bit time and **_timeb** contains a 32-bit time.

_ftime validates its parameters. If passed a null pointer as *timeptr*, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL**.

Requirements

FUNCTION	REQUIRED HEADER
_ftime	<sys/types.h> and <sys/timeb.h>
_ftime32	<sys/types.h> and <sys/timeb.h>
_ftime64	<sys/types.h> and <sys/timeb.h>

For more compatibility information, see Compatibility.

Example

```
// crt_ftime.c
// compile with: /W3
// This program uses _ftime to obtain the current
// time and then stores this time in timebuffer.

#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>

int main( void )
{
   struct _timeb timebuffer;
   char timeline[26];
   errno_t err;
   time_t time1;
   unsigned short millitm1;
   short timezone1;
   short dstflag1;

   _ftime( &timebuffer ); // C4996
   // Note: _ftime is deprecated; consider using _ftime_s instead

   time1 = timebuffer.time;
   millitm1 = timebuffer.millitm;
   timezone1 = timebuffer.timezone;
   dstflag1 = timebuffer.dstflag;

   printf( "Seconds since midnight, January 1, 1970 (UTC): %I64d\n",
   time1);
   printf( "Milliseconds: %d\n", millitm1);
   printf( "Minutes between UTC and local time: %d\n", timezone1);
   printf( "Daylight savings time flag (1 means Daylight time is in "
           "effect): %d\n", dstflag1);

   err = ctime_s( timeline, 26, & ( timebuffer.time ) );
   if (err)
   {
       printf("Invalid argument to ctime_s. ");
   }
   printf( "The time is %.19s.%hu %s", timeline, timebuffer.millitm,
           &timeline[20] );
}
```

```
Seconds since midnight, January 1, 1970 (UTC): 1051553334
Milliseconds: 230
Minutes between UTC and local time: 480
Daylight savings time flag (1 means Daylight time is in effect): 1
The time is Mon Apr 28 11:08:54.230 2003
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

_ftime_s, _ftime32_s, _ftime64_s

10/31/2018 • 2 minutes to read • Edit Online

Gets the current time. These are versions of _ftime, _ftime32, _ftime64 with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _ftime_s( struct _timeb *timeptr );
errno_t _ftime32_s( struct __timeb32 *timeptr );
errno_t _ftime64_s( struct __timeb64 *timeptr );
```

Parameters

timeptr
Pointer to a **_timeb**, **__timeb32**, or **__timeb64** structure.

Return Value

Zero if successful, an error code on failure. If *timeptr* is **NULL**, the return value is **EINVAL**.

Remarks

The **_ftime_s** function gets the current local time and stores it in the structure pointed to by *timeptr*. The **_timeb**, **__timeb32**, and **__timeb64** structures are defined in SYS\Timeb.h. They contain four fields, which are listed in the following table.

FIELD	DESCRIPTION
dstflag	Nonzero if daylight savings time is currently in effect for the local time zone. (See _tzset for an explanation of how daylight savings time is determined.)
millitm	Fraction of a second in milliseconds.
time	Time in seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).
timezone	Difference in minutes, moving westward, between UTC and local time. The value of **timezone** is set from the value of the global variable **_timezone** (see **_tzset**).

The **_ftime64_s** function, which uses the **__timeb64** structure, allows file-creation dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **_ftime32_s** only represents dates through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for all these functions.

The **_ftime_s** function is equivalent to **_ftime64_s**, and **_timeb** contains a 64-bit time, unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_ftime_s** uses a 32-bit time and **_timeb** contains a 32-bit time.

_ftime_s validates its parameters. If passed a null pointer as *timeptr*, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to

EINVAL.

Requirements

FUNCTION	REQUIRED HEADER
_ftime_s	<sys/types.h> and <sys/timeb.h>
_ftime32_s	<sys/types.h> and <sys/timeb.h>
_ftime64_s	<sys/types.h> and <sys/timeb.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_ftime64_s.c
// This program uses _ftime64_s to obtain the current
// time and then stores this time in timebuffer.

#include <stdio.h>
#include <sys/timeb.h>
#include <time.h>

int main( void )
{
   struct _timeb timebuffer;
   char timeline[26];
   errno_t err;
   time_t time1;
   unsigned short millitm1;
   short timezone1;
   short dstflag1;

   _ftime64_s( &timebuffer );

    time1 = timebuffer.time;
    millitm1 = timebuffer.millitm;
    timezone1 = timebuffer.timezone;
    dstflag1 = timebuffer.dstflag;

   printf( "Seconds since midnight, January 1, 1970 (UTC): %I64d\n",
   time1);
   printf( "Milliseconds: %d\n", millitm1);
   printf( "Minutes between UTC and local time: %d\n", timezone1);
   printf( "Daylight savings time flag (1 means Daylight time is in "
           "effect): %d\n", dstflag1);

   err = ctime_s( timeline, 26, & ( timebuffer.time ) );
   if (err)
   {
       printf("Invalid argument to ctime_s. ");
   }
   printf( "The time is %.19s.%hu %s", timeline, timebuffer.millitm,
           &timeline[20] );
}
```

```
Seconds since midnight, January 1, 1970 (UTC): 1051553334
Milliseconds: 230
Minutes between UTC and local time: 480
Daylight savings time flag (1 means Daylight time is in effect): 1
The time is Mon Apr 28 11:08:54.230 2003
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64

_fullpath, _wfullpath

10/31/2018 • 3 minutes to read • Edit Online

Creates an absolute or full path name for the specified relative path name.

Syntax

```
char *_fullpath(
   char *absPath,
   const char *relPath,
   size_t maxLength
);
wchar_t *_wfullpath(
   wchar_t *absPath,
   const wchar_t *relPath,
   size_t maxLength
);
```

Parameters

absPath
Pointer to a buffer containing the absolute or full path name, or **NULL**.

relPath
Relative path name.

maxLength
Maximum length of the absolute path name buffer (*absPath*). This length is in bytes for **_fullpath** but in wide characters (**wchar_t**) for **_wfullpath**.

Return Value

Each of these functions returns a pointer to a buffer containing the absolute path name (*absPath*). If there is an error (for example, if the value passed in *relPath* includes a drive letter that is not valid or cannot be found, or if the length of the created absolute path name (*absPath*) is greater than *maxLength*), the function returns **NULL**.

Remarks

The **_fullpath** function expands the relative path name in *relPath* to its fully qualified or absolute path and stores this name in *absPath*. If *absPath* is **NULL**, **malloc** is used to allocate a buffer of sufficient length to hold the path name. It is the responsibility of the caller to free this buffer. A relative path name specifies a path to another location from the current location (such as the current working directory: "."). An absolute path name is the expansion of a relative path name that states the entire path required to reach the desired location from the root of the file system. Unlike **_makepath**, **_fullpath** can be used to obtain the absolute path name for relative paths (*relPath*) that include "./" or "../" in their names.

For example, to use C run-time routines, the application must include the header files that contain the declarations for the routines. Each header file include statement references the location of the file in a relative manner (from the application's working directory):

```
#include <stdlib.h>
```

when the absolute path (actual file system location) of the file might be:

```
\\machine\shareName\msvcSrc\crt\headerFiles\stdlib.h
```

_fullpath automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wfullpath** is a wide-character version of **_fullpath**; the string arguments to **_wfullpath** are wide-character strings. **_wfullpath** and **_fullpath** behave identically except that **_wfullpath** does not handle multibyte-character strings.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are both defined, calls to **_fullpath** and **_wfullpath** are replaced by calls to **_fullpath_dbg** and **_wfullpath_dbg** to allow for debugging memory allocations. For more information, see _fullpath_dbg, _wfullpath_dbg.

This function invokes the invalid parameter handler, as described in Parameter Validation, if *maxlen* is less than or equal to 0. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfullpath	**_fullpath**	**_fullpath**	**_wfullpath**

If the *absPath* buffer is **NULL**, **_fullpath** calls malloc to allocate a buffer and ignores the *maxLength* argument. It is the caller's responsibility to deallocate this buffer (using free) as appropriate. If the *relPath* argument specifies a disk drive, the current directory of this drive is combined with the path.

Requirements

FUNCTION	REQUIRED HEADER
_fullpath	<stdlib.h>
_wfullpath	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_fullpath.c
// This program demonstrates how _fullpath
// creates a full path from a partial path.

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <direct.h>

void PrintFullPath( char * partialPath )
{
   char full[_MAX_PATH];
   if( _fullpath( full, partialPath, _MAX_PATH ) != NULL )
      printf( "Full path is: %s\n", full );
   else
      printf( "Invalid path\n" );
}

int main( void )
{
   PrintFullPath( "test" );
   PrintFullPath( "\\test" );
   PrintFullPath( "..\\test" );
}
```

```
Full path is: C:\Documents and Settings\user\My Documents\test
Full path is: C:\test
Full path is: C:\Documents and Settings\user\test
```

See also

File Handling
_getcwd, _wgetcwd
_getdcwd, _wgetdcwd
_makepath, _wmakepath
_splitpath, _wsplitpath

_fullpath_dbg, _wfullpath_dbg

10/31/2018 • 2 minutes to read • Edit Online

Versions of _fullpath, _wfullpath that use the debug version of **malloc** to allocate memory.

Syntax

```
char *_fullpath_dbg(
   char *absPath,
   const char *relPath,
   size_t maxLength,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wfullpath_dbg(
   wchar_t *absPath,
   const wchar_t *relPath,
   size_t maxLength,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

absPath
Pointer to a buffer containing the absolute or full path name, or **NULL**.

relPath
Relative path name.

maxLength
Maximum length of the absolute path name buffer (*absPath*). This length is in bytes for **_fullpath** but in wide characters (**wchar_t**) for **_wfullpath**.

blockType
Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested allocation operation or **NULL**.

linenumber
Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Each function returns a pointer to a buffer containing the absolute path name (*absPath*). If there is an error (for example, if the value passed in *relPath* includes a drive letter that is not valid or cannot be found, or if the length of the created absolute path name (*absPath*) is greater than *maxLength*) the function returns **NULL**.

Remarks

The **_fullpath_dbg** and **_wfullpath_dbg** functions are identical to **_fullpath** and **_wfullpath** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc**, **_malloc_dbg**, to allocate memory if **NULL**

is passed as the first parameter. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_fullpath** and **_wfullpath** are remapped to **_fullpath_dbg** and **_wfullpath_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of blocks on the debug heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tfullpath_dbg	**_fullpath_dbg**	**_fullpath_dbg**	**_wfullpath_dbg**

Requirements

FUNCTION	REQUIRED HEADER
_fullpath_dbg	<crtdbg.h>
_wfullpath_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

See also

File Handling
_fullpath, _wfullpath
Debug Versions of Heap Allocation Functions

_futime, _futime32, _futime64

10/31/2018 • 2 minutes to read • Edit Online

Sets the modification time on an open file.

Syntax

```
int _futime(
   int fd,
   struct _utimbuf *filetime
);
int _futime32(
   int fd,
   struct __utimbuf32 *filetime
);
int _futime64(
   int fd,
   struct __utimbuf64 *filetime
);
```

Parameters

fd
File descriptor to the open file.

filetime
Pointer to the structure containing the new modification date.

Return Value

Return 0 if successful. If an error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to **EBADF**, indicating an invalid file descriptor, or **EINVAL**, indicating an invalid parameter.

Remarks

The **_futime** routine sets the modification date and the access time on the open file associated with *fd*. **_futime** is identical to _utime, except that its argument is the file descriptor of an open file, rather than the name of a file or a path to a file. The **_utimbuf** structure contains fields for the new modification date and access time. Both fields must contain valid values. **_utimbuf32** and **_utimbuf64** are identical to **_utimbuf** except for the use of the 32-bit and 64-bit time types, respectively. **_futime** and **_utimbuf** use a 64-bit time type and **_futime** is identical in behavior to **_futime64**. If you need to force the old behavior, define **_USE_32BIT_TIME_T**. Doing this causes **_futime** to be identical in behavior to **_futime32** and causes the **_utimbuf** structure to use the 32-bit time type, making it equivalent to **__utimbuf32**.

_futime64, which uses the **__utimbuf64** structure, can read and modify file dates through 23:59:59, December 31, 3000, UTC; whereas a call to **_futime32** fails if the date on the file is later than 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for these functions.

Requirements

FUNCTION	REQUIRED HEADER	OPTIONAL HEADER
_futime	<sys/utime.h>	<errno.h>
_futime32	<sys/utime.h>	<errno.h>
_futime64	<sys/utime.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_futime.c
// This program uses _futime to set the
// file-modification time to the current time.

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/utime.h>
#include <share.h>

int main( void )
{
   int hFile;

   // Show file time before and after.
   system( "dir crt_futime.c_input" );

   _sopen_s( &hFile, "crt_futime.c_input", _O_RDWR, _SH_DENYNO, 0 );

   if( _futime( hFile, NULL ) == -1 )
      perror( "_futime failed\n" );
   else
      printf( "File time modified\n" );

   _close (hFile);

   system( "dir crt_futime.c_input" );
}
```

Input: crt_futime.c_input

```
Arbitrary file contents.
```

Sample Output

```
Volume in drive Z has no label.
Volume Serial Number is 5C68-57C1

Directory of Z:\crt

03/25/2004  10:40 AM                24 crt_futime.c_input
               1 File(s)             24 bytes
               0 Dir(s)  24,268,476,416 bytes free
Volume in drive Z has no label.
Volume Serial Number is 5C68-57C1

Directory of Z:\crt

03/25/2004  10:41 AM                24 crt_futime.c_input
               1 File(s)             24 bytes
               0 Dir(s)  24,268,476,416 bytes free
File time modified
```

See also

Time Management

fwide

10/31/2018 • 2 minutes to read • Edit Online

Unimplemented.

Syntax

```
int fwide(
   FILE *stream,
   int mode;
);
```

Parameters

stream
Pointer to **FILE** structure (ignored).

mode
The new width of the stream: positive for wide character, negative for byte, zero to leave unchanged. (This value is ignored.)

Return Value

This function currently just returns *mode*.

Remarks

The current version of this function does not comply with the Standard.

Requirements

FUNCTION	REQUIRED HEADER
fwide	<wchar.h>

For more information, see Compatibility.

fwrite

10/31/2018 • 2 minutes to read • Edit Online

Writes data to a stream.

Syntax

```
size_t fwrite(
   const void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer
Pointer to data to be written.

size
Item size, in bytes.

count
Maximum number of items to be written.

stream
Pointer to **FILE** structure.

Return Value

fwrite returns the number of full items actually written, which may be less than *count* if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined. If either *stream* or *buffer* is a null pointer, or if an odd number of bytes to be written is specified in Unicode mode, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns 0.

Remarks

The **fwrite** function writes up to *count* items, of *size* length each, from *buffer* to the output *stream*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually written. If *stream* is opened in text mode, each linefeed is replaced with a carriage-return - linefeed pair. The replacement has no effect on the return value.

When *stream* is opened in Unicode translation mode—for example, if *stream* is opened by calling **fopen** and using a mode parameter that includes **ccs=UNICODE**, **ccs=UTF-16LE**, or **ccs=UTF-8**, or if the mode is changed to a Unicode translation mode by using **_setmode** and a mode parameter that includes **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**—*buffer* is interpreted as a pointer to an array of **wchar_t** that contains UTF-16 data. An attempt to write an odd number of bytes in this mode causes a parameter validation error.

Because this function locks the calling thread, it is thread-safe. For a non-locking version, see **_fwrite_nolock**.

Requirements

FUNCTION	REQUIRED HEADER
fwrite	<stdio.h>

For additional compatibility information, see Compatibility.

Example

See the example for fread.

See also

Stream I/O
_setmode
fread
_fwrite_nolock
_write

_fwrite_nolock

10/31/2018 • 2 minutes to read • Edit Online

Writes data to a stream, without locking the thread.

Syntax

```
size_t _fwrite_nolock(
   const void *buffer,
   size_t size,
   size_t count,
   FILE *stream
);
```

Parameters

buffer
Pointer to the data to be written.

size
Item size in bytes.

count
Maximum number of items to be written.

stream
Pointer to the **FILE** structure.

Return Value

Same as fwrite.

Remarks

This function is a non-locking version of **fwrite**. It is identical to **fwrite** except that it is not protected from interference by other threads. It might be faster because it does not incur the overhead of locking out other threads. Use this function only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Requirements

FUNCTION	REQUIRED HEADER
_fwrite_nolock	<stdio.h>

For more compatibility information, see Compatibility.

Example

See the example for fread.

See also

Stream I/O
fread
_write

gcvt

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _gcvt or security-enhanced _gcvt_s instead.

_gcvt

10/31/2018 • 2 minutes to read • Edit Online

Converts a floating-point value to a string, which it stores in a buffer. A more secure version of this function is available; see _gcvt_s.

Syntax

```
char *_gcvt(
   double value,
   int digits,
   char *buffer
);
```

Parameters

value
Value to be converted.

digits
Number of significant digits stored.

buffer
Storage location for the result.

Return Value

_gcvt returns a pointer to the string of digits.

Remarks

The **_gcvt** function converts a floating-point *value* to a character string (which includes a decimal point and a possible sign byte) and stores the string in *buffer*. The *buffer* should be large enough to accommodate the converted value plus a terminating null character, which is appended automatically. If a buffer size of *digits* + 1 is used, the function overwrites the end of the buffer. This is because the converted string includes a decimal point and can contain sign and exponent information. There is no provision for overflow. **_gcvt** attempts to produce *digits* digits in decimal format. If it cannot, it produces *digits* digits in exponential format. Trailing zeros might be suppressed in the conversion.

A *buffer* of length **_CVTBUFSIZE** is sufficient for any floating point value.

This function validates its parameters. If *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_gcvt	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_gcvt.c
// compile with: /W3
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
   char buffer[_CVTBUFSIZE];
   double value = -1234567890.123;
   printf( "The following numbers were converted by _gcvt(value,12,buffer):\n" );
   _gcvt( value, 12, buffer ); // C4996
   // Note: _gcvt is deprecated; consider using _gcvt_s instead
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value *= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value *= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value *= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );

   printf( "\n" );
   value = -12.34567890123;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value /= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value /= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
   value /= 10;
   _gcvt( value, 12, buffer ); // C4996
   printf( "buffer: '%s' (%d chars)\n", buffer, strlen(buffer) );
}
```

```
The following numbers were converted by _gcvt(value,12,buffer):
buffer: '-1234567890.12' (14 chars)
buffer: '-12345678901.2' (14 chars)
buffer: '-123456789012' (13 chars)
buffer: '-1.23456789012e+012' (19 chars)

buffer: '-12.3456789012' (14 chars)
buffer: '-1.23456789012' (14 chars)
buffer: '-0.123456789012' (15 chars)
buffer: '-1.23456789012e-002' (19 chars)
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt
_fcvt

_gcvt_s

10/31/2018 • 2 minutes to read • Edit Online

Converts a floating-point value to a string. This is a version of _gcvt with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _gcvt_s(
   char *buffer,
   size_t sizeInBytes,
   double value,
   int digits
);
template <size_t cchStr>
errno_t _gcvt_s(
   char (&buffer)[cchStr],
   double value,
   int digits
); // C++ only
```

Parameters

buffer
Buffer to store the result of the conversion.

sizeInBytes
Size of the buffer.

value
Value to be converted.

digits
Number of significant digits stored.

Return Value

Zero if successful. If a failure occurs due to an invalid parameter (see the following table for invalid values), the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, an error code is returned. Error codes are defined in Errno.h. For a listing of these errors, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

BUFFER	*SIZEINBYTES*	*VALUE*	*DIGITS*	RETURN	VALUE IN *BUFFER*
NULL	any	any	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	zero	any	any	**EINVAL**	Not modified.

BUFFER	*SIZEINBYTES*	*VALUE*	*DIGITS*	RETURN	VALUE IN *BUFFER*
Not **NULL** (points to valid memory)	any	any	>= *sizeInBytes*	**EINVAL**	Not modified.

Security Issues

_gcvt_s can generate an access violation if *buffer* does not point to valid memory and is not **NULL**.

Remarks

The **_gcvt_s** function converts a floating-point *value* to a character string (which includes a decimal point and a possible sign byte) and stores the string in *buffer*. *buffer* should be large enough to accommodate the converted value plus a terminating null character, which is appended automatically. A buffer of length **_CVTBUFSIZE** is sufficient for any floating point value. If a buffer size of *digits* + 1 is used, the function will not overwrite the end of the buffer, so be sure to supply a sufficient buffer for this operation. **_gcvt_s** attempts to produce *digits* digits in decimal format. If it cannot, it produces *digits* digits in exponential format. Trailing zeros can be suppressed in the conversion.

In C++, using this function is simplified by a template overload; the overload can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug version of this function first fills the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_gcvt_s	<stdlib.h>	<error.h>

For more compatibility information, see Compatibility.

Example

```
// crt_gcvt_s.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    char buf[_CVTBUFSIZE];
    int decimal;
    int sign;
    int err;

    err = _gcvt_s(buf, _CVTBUFSIZE, 1.2, 5);

    if (err != 0)
    {
        printf("_gcvt_s failed with error code %d\n", err);
        exit(1);
    }

    printf("Converted value: %s\n", buf);
}
```

```
Converted value: 1.2
```

See also

Data Conversion
Floating-Point Support
atof, _atof_l, _wtof, _wtof_l
_ecvt_s
_fcvt_s
_gcvt

_get_current_locale

10/31/2018 • 2 minutes to read • Edit Online

Gets a locale object representing the current locale.

Syntax

```
_locale_t _get_current_locale(void);
```

Return Value

A locale object representing the current locale.

Remarks

The **_get_current_locale** function gets the currently set locale for the thread and returns a locale object representing that locale.

The previous name of this function, **__get_current_locale** (with two leading underscores) has been deprecated.

Requirements

ROUTINE	REQUIRED HEADER
_get_current_locale	<locale.h>

For more compatibility information, see Compatibility.

See also

setlocale, _wsetlocale
_create_locale, _wcreate_locale
_free_locale

_get_daylight

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the daylight saving time offset in hours.

Syntax

```
error_t _get_daylight( int* hours );
```

Parameters

hours
The offset in hours of daylight saving time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_daylight** function retrieves the number of hours in daylight saving time as an integer. If daylight saving time is in effect, the default offset is one hour (although a few regions do observe a two-hour offset).

If *hours* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

We recommend you use this function instead of the macro **_daylight** or the deprecated function **__daylight**.

Requirements

ROUTINE	REQUIRED HEADER
_get_daylight	<time.h>

For more information, see Compatibility.

See also

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_dstbias
_get_timezone
_get_tzname

_get_doserrno

11/8/2018 • 2 minutes to read • Edit Online

Gets the error value returned by the operating system before it is translated into an **errno** value.

Syntax

```
errno_t _get_doserrno(
   int * pValue
);
```

Parameters

pValue
A pointer to an integer to be filled with the current value of the **_doserrno** global macro.

Return Value

If **_get_doserrno** succeeds, it returns zero; if it fails, it returns an error code. If *pValue* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The **_doserrno** global macro is set to zero during CRT initialization, before process execution begins. It is set to the operating-system error value returned by any system-level function call that returns an operating-system error, and it is never reset to zero during execution. When you write code to check the error value returned by a function, always clear **_doserrno** by using _set_doserrno before the function call. Because another function call may overwrite **_doserrno**, check the value by using **_get_doserrno** immediately after the function call.

We recommend _get_errno instead of **_get_doserrno** for portable error codes.

Possible values of **_doserrno** are defined in <errno.h>.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_get_doserrno	<stdlib.h>, <cstdlib> (C++)	<errno.h>, <cerrno> (C++)

_get_doserrno is a Microsoft extension. For more compatibility information, see Compatibility.

See also

_set_doserrno
errno, _doserrno, _sys_errlist, and _sys_nerr

_get_dstbias

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the daylight saving time offset in seconds.

Syntax

```
error_t _get_dstbias( int* seconds );
```

Parameters

seconds
The offset in seconds of daylight saving time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_dstbias** function retrieves the number of seconds in daylight saving time as an integer. If daylight saving time is in effect, the default offset is 3600 seconds, which is the number of seconds in one hour (though a few regions do observe a two-hour offset).

If *seconds* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

We recommend you use this function instead of the macro **_dstbias** or the deprecated function **__dstbias**.

Requirements

ROUTINE	REQUIRED HEADER
_get_dstbias	<time.h>

For more information, see Compatibility.

See also

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_timezone
_get_tzname

_get_errno

11/8/2018 • 2 minutes to read • Edit Online

Gets the current value of the errno global variable.

Syntax

```
errno_t _get_errno(
   int * pValue
);
```

Parameters

pValue
A pointer to an integer to be filled with the current value of the **errno** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

Possible values of **errno** are defined in Errno.h. Also, see errno Constants.

Example

```
// crt_get_errno.c
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <share.h>
#include <errno.h>

int main()
{
   errno_t err;
   int pfh;
   _sopen_s( &pfh, "nonexistent.file", _O_WRONLY, _SH_DENYNO, _S_IWRITE );
   _get_errno( &err );
   printf( "errno = %d\n", err );
   printf( "fyi, ENOENT = %d\n", ENOENT );
}
```

```
errno = 2
fyi, ENOENT = 2
```

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_get_errno	<stdlib.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

_set_errno
errno, _doserrno, _sys_errlist, and _sys_nerr

_get_FMA3_enable, _set_FMA3_enable

10/31/2018 • 2 minutes to read • Edit Online

Gets or sets a flag that specifies whether the transcendental math floating-point library functions use FMA3 instructions in code compiled for X64 platforms.

Syntax

```
int _set_FMA3_enable(int flag);
int _get_FMA3_enable();
```

Parameters

flag
Set to 1 to enable the FMA3 implementations of the transcendental math floating-point library functions on X64 platforms, or to 0 to use the implementations that do not use FMA3 instructions.

Return Value

A non-zero value if the FMA3 implementations of the transcendental math floating-point library functions are enabled. Otherwise, zero.

Remarks

Use the **_set_FMA3_enable** function to enable or disable the use of FMA3 instructions in the transcendental math floating-point functions in the CRT library. The return value reflects the implementation in use after the change. If the CPU does not support FMA3 instructions, this function cannot enable them in the library, and the return value is zero. Use **_get_FMA3_enable** to get the current state of the library. By default, on X64 platforms, the CRT startup code detects whether the CPU supports FMA3 instructions, and enables or disables the FMA3 implementations in the library.

Because the FMA3 implementations use different algorithms, slight differences in the result of computations may be observable when the FMA3 implementations are enabled or disabled, or between computers that do or do not support FMA3. For more information, see Floating-point migration issues.

Requirements

The **_set_FMA3_enable** and **_get_FMA3_enable** functions are only available in the X64 versions of the CRT.

ROUTINE	REQUIRED HEADER
_set_FMA3_enable, **_get_FMA3_enable**	C: <math.h> C++: <cmath> or <math.h>

The **_set_FMA3_enable** and **_get_FMA3_enable** functions are Microsoft specific. For compatibility information, see Compatibility.

See also

Floating-point support

_get_fmode

11/8/2018 • 2 minutes to read • Edit Online

Gets the default file translation mode for file I/O operations.

Syntax

```
errno_t _get_fmode(
   int * pmode
);
```

Parameters

pmode
A pointer to an integer to be filled with the current default mode: **_O_TEXT** or **_O_BINARY**.

Return Value

Returns zero if successful; an error code on failure. If *pmode* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EINVAL**.

Remarks

The function gets the value of the _fmode global variable. This variable specifies the default file translation mode for both low-level and stream file I/O operations, such as **_open**, **_pipe**, **fopen**, and freopen.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_get_fmode	<stdlib.h>	<fcntl.h>

For more compatibility information, see Compatibility.

Example

See the example in _set_fmode.

See also

_fmode
_set_fmode
_setmode
Text and Binary Mode File I/O

_get_heap_handle

10/31/2018 • 2 minutes to read • Edit Online

Returns the handle of the heap that's used by the C run-time system.

Syntax

```
intptr_t _get_heap_handle( void );
```

Return Value

Returns the handle to the Win32 heap used by the C run-time system.

Remarks

Use this function if you want to call HeapSetInformation and enable the Low Fragmentation Heap on the CRT heap.

Requirements

ROUTINE	REQUIRED HEADER
_get_heap_handle	<malloc.h>

For more compatibility information, see Compatibility.

Sample

```
// crt_get_heap_handle.cpp
// compile with: /MT
#include <windows.h>
#include <malloc.h>
#include <stdio.h>

int main(void)
{
    intptr_t hCrtHeap = _get_heap_handle();
    ULONG ulEnableLFH = 2;
    if (HeapSetInformation((PVOID)hCrtHeap,
                           HeapCompatibilityInformation,
                           &ulEnableLFH, sizeof(ulEnableLFH)))
        puts("Enabling Low Fragmentation Heap succeeded");
    else
        puts("Enabling Low Fragmentation Heap failed");
    return 0;
}
```

See also

Memory Allocation

_get_invalid_parameter_handler, _get_thread_local_invalid_parameter_handler

10/31/2018 • 2 minutes to read • Edit Online

Gets the function that is called when the CRT detects an invalid argument.

Syntax

```
_invalid_parameter_handler _get_invalid_parameter_handler(void);
_invalid_parameter_handler _get_thread_local_invalid_parameter_handler(void);
```

Return Value

A pointer to the currently set invalid parameter handler function, or a null pointer if none has been set.

Remarks

The **_get_invalid_parameter_handler** function gets the currently set global invalid parameter handler. It returns a null pointer if no global invalid parameter handler was set. Similarly, the **_get_thread_local_invalid_parameter_handler** gets the current thread-local invalid parameter handler of the thread it is called on, or a null pointer if no handler was set. For information about how to set global and thread-local invalid parameter handlers, see _set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler.

The returned invalid parameter handler function pointer has the following type:

```
typedef void (__cdecl* _invalid_parameter_handler)(
    wchar_t const*,
    wchar_t const*,
    wchar_t const*,
    unsigned int,
    uintptr_t
    );
```

For details on the invalid parameter handler, see the prototype in _set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler.

Requirements

ROUTINE	REQUIRED HEADER
_get_invalid_parameter_handler, **_get_thread_local_invalid_parameter_handler**	C: <stdlib.h> C++: <cstdlib> or <stdlib.h>

The **_get_invalid_parameter_handler** and **_get_thread_local_invalid_parameter_handler** functions are Microsoft specific. For compatibility information, see Compatibility.

See also

_set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler
Security-Enhanced Versions of CRT Functions

_get_osfhandle

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the operating-system file handle that is associated with the specified file descriptor.

Syntax

```
intptr_t _get_osfhandle(
   int fd
);
```

Parameters

fd
An existing file descriptor.

Return Value

Returns an operating-system file handle if *fd* is valid. Otherwise, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **INVALID_HANDLE_VALUE** (-1) and sets **errno** to **EBADF**, indicating an invalid file handle. To avoid a compiler warning when the result is used in routines that expect a Win32 file handle, cast it to a **HANDLE** type.

Remarks

To close a file whose operating system (OS) file handle is obtained by **_get_osfhandle**, call _close on the file descriptor *fd*. Do not call **CloseHandle** on the return value of this function. The underlying OS file handle is owned by the *fd* file descriptor, and is closed when _close is called on *fd*. If the file descriptor is owned by a `FILE *` stream, then calling fclose on that `FILE *` stream closes both the file descriptor and the underlying OS file handle. In this case, do not call _close on the file descriptor.

Requirements

ROUTINE	REQUIRED HEADER
_get_osfhandle	<io.h>

For more compatibility information, see Compatibility.

See also

File Handling
_close
_creat, _wcreat
_dup, _dup2
_open, _wopen

_get_pgmptr

11/8/2018 • 2 minutes to read • Edit Online

Gets the current value of the **_pgmptr** global variable.

Syntax

```
errno_t _get_pgmptr(
   char **pValue
);
```

Parameters

pValue
A pointer to a string to be filled with the current value of the **_pgmptr** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

Only call **_get_pgmptr** if your program has a narrow entry point, like **main()** or **WinMain()**. The **_pgmptr** global variable contains the full path to the executable associated with the process. For more information, see _pgmptr, _wpgmptr.

Requirements

ROUTINE	REQUIRED HEADER
_get_pgmptr	<stdlib.h>

For more compatibility information, see Compatibility.

See also

_get_wpgmptr

_get_printf_count_output

10/31/2018 • 2 minutes to read • Edit Online

Indicates whether printf, _printf_l, wprintf, _wprintf_l-family functions support the **%n** format.

Syntax

```
int _get_printf_count_output();
```

Return Value

Non-zero if **%n** is supported, 0 if **%n** is not supported.

Remarks

If **%n** is not supported (the default), encountering **%n** in the format string of any of the **printf** functions will invoke the invalid parameter handler as described in Parameter Validation. If **%n** support is enabled (see _set_printf_count_output) then **%n** will behave as described in Format Specification Syntax: printf and wprintf Functions.

Requirements

ROUTINE	REQUIRED HEADER
_get_printf_count_output	<stdio.h>

For additional compatibility information, see Compatibility.

Example

See the example for _set_printf_count_output.

See also

_set_printf_count_output

_get_purecall_handler, _set_purecall_handler

10/31/2018 • 2 minutes to read • Edit Online

Gets or sets the error handler for a pure virtual function call.

Syntax

```
typedef void (__cdecl* _purecall_handler)(void);
_purecall_handler __cdecl _get_purecall_handler(void);
_purecall_handler __cdecl _set_purecall_handler(
   _purecall_handler function
);
```

Parameters

function
The function to be called when a pure virtual function is called. A **_purecall_handler** function must have a void return type.

Return Value

The previous **_purecall_handler**. Returns **nullptr** if there was no previous handler.

Remarks

The **_get_purecall_handler** and **_set_purecall_handler** functions are Microsoft-specific and apply only to C++ code.

A call to a pure virtual function is an error because it has no implementation. By default, the compiler generates code to invoke an error handler function when a pure virtual function is called, which terminates the program. You can install your own error handler function for pure virtual function calls, to catch them for debugging or reporting purposes. To use your own error handler, create a function that has the **_purecall_handler** signature, then use **_set_purecall_handler** to make it the current handler.

Because there is only one **_purecall_handler** for each process, when you call **_set_purecall_handler** it immediately impacts all threads. The last caller on any thread sets the handler.

To restore the default behavior, call **_set_purecall_handler** by using a **nullptr** argument.

Requirements

ROUTINE	REQUIRED HEADER
_get_purecall_handler, **_set_purecall_handler**	<cstdlib> or <stdlib.h>

For compatibility information, see Compatibility.

Example

```
// _set_purecall_handler.cpp
// compile with: /W1
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>

class CDerived;
class CBase
{
public:
   CBase(CDerived *derived): m_pDerived(derived) {};
   ~CBase();
   virtual void function(void) = 0;

   CDerived * m_pDerived;
};

class CDerived : public CBase
{
public:
   CDerived() : CBase(this) {};   // C4355
   virtual void function(void) {};
};

CBase::~CBase()
{
   m_pDerived -> function();
}

void myPurecallHandler(void)
{
   printf("In _purecall_handler.");
   exit(0);
}

int _tmain(int argc, _TCHAR* argv[])
{
   _set_purecall_handler(myPurecallHandler);
   CDerived myDerived;
}
```

```
In _purecall_handler.
```

See also

Error Handling

_purecall

_get_terminate

10/31/2018 • 2 minutes to read • Edit Online

Returns the termination routine to be called by **terminate**.

Syntax

```
terminate_function _get_terminate( void );
```

Return Value

Returns a pointer to the function registered by set_terminate. If no function has been set, the return value may be used to restore the default behavior; this value may be **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_get_terminate	<eh.h>

For additional compatibility information, see Compatibility.

See also

Exception Handling Routines
abort
set_unexpected
terminate
unexpected

_get_timezone

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the difference in seconds between coordinated universal time (UTC) and local time.

Syntax

```
error_t _get_timezone(
    long* seconds
);
```

Parameters

seconds
The difference in seconds between UTC and local time.

Return Value

Zero if successful or an **errno** value if an error occurs.

Remarks

The **_get_timezone** function retrieves the difference in seconds between UTC and local time as an integer. The default value is 28,800 seconds, for Pacific Standard Time (eight hours behind UTC).

If *seconds* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_get_timezone	<time.h>

For more information, see Compatibility.

See also

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_dstbias
_get_tzname

_get_tzname

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the character string representation of the time zone name or the daylight standard time zone name (DST).

Syntax

```
errno_t _get_tzname(
    size_t* pReturnValue,
    char* timeZoneName,
    size_t sizeInBytes,
    int index
);
```

Parameters

pReturnValue
The string length of *timeZoneName* including a null terminator.

timeZoneName
The address of a character string for the representation of the time zone name or the daylight standard time zone name (DST), depending on *index*.

sizeInBytes
The size of the *timeZoneName* character string in bytes.

index
The index of one of the two time zone names to retrieve.

INDEX	CONTENTS OF *TIMEZONENAME*	*TIMEZONENAME* DEFAULT VALUE
0	Time zone name	"PST"
1	Daylight standard time zone name	"PDT"
> 1 or < 0	**errno** set to **EINVAL**	not modified

Unless the values are explicitly changed during run time, the default values are "PST" and "PDT" respectively.

Return Value

Zero if successful, otherwise an **errno** type value.

If either *timeZoneName* is **NULL**, or *sizeInBytes* is zero or less than zero (but not both), an invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Error Conditions

PRETURNVALUE	*TIMEZONENAME*	*SIZEINBYTES*	*INDEX*	RETURN VALUE	CONTENTS OF *TIMEZONENAME*
size of TZ name	**NULL**	0	0 or 1	0	not modified
size of TZ name	any	> 0	0 or 1	0	TZ name
not modified	**NULL**	> 0	any	**EINVAL**	not modified
not modified	any	zero	any	**EINVAL**	not modified
not modified	any	> 0	> 1	**EINVAL**	not modified

Remarks

The **_get_tzname** function retrieves the character string representation of the current time zone name or the daylight standard time zone name (DST) into the address of *timeZoneName* depending on the index value, along with the size of the string in *pReturnValue*. If *timeZoneName* is **NULL** and *sizeInBytes* is zero, the size of the string required to hold the specified time zone and a terminating null in bytes is returned in *pReturnValue*. The index values must be either 0 for standard time zone or 1 for daylight standard time zone; any other values of *index* have undetermined results.

Example

This sample calls **_get_tzname** to get the required buffer size to display the current Daylight standard time zone name, allocates a buffer of that size, calls **_get_tzname** again to load the name in the buffer, and prints it to the console.

```
// crt_get_tzname.c
// Compile by using: cl /W4 crt_get_tzname.c
#include <stdio.h>
#include <time.h>
#include <malloc.h>

enum TZINDEX {
    STD,
    DST
};

int main()
{
    size_t tznameSize = 0;
    char * tznameBuffer = NULL;

    // Get the size of buffer required to hold DST time zone name
    if (_get_tzname(&tznameSize, NULL, 0, DST))
        return 1;    // Return an error value if it failed

    // Allocate a buffer for the name
    if (NULL == (tznameBuffer = (char *)(malloc(tznameSize))))
        return 2;    // Return an error value if it failed

    // Load the name in the buffer
    if (_get_tzname(&tznameSize, tznameBuffer, tznameSize, DST))
        return 3;    // Return an error value if it failed

    printf_s("The current Daylight standard time zone name is %s.\n", tznameBuffer);
    return 0;
}
```

Output

```
The current Daylight standard time zone name is PDT.
```

Requirements

ROUTINE	REQUIRED HEADER
_get_tzname	<time.h>

For more information, see Compatibility.

See also

Time Management
errno, _doserrno, _sys_errlist, and _sys_nerr
_get_daylight
_get_dstbias
_get_timezone

_get_unexpected

10/31/2018 • 2 minutes to read • Edit Online

Returns the termination routine to be called by **unexpected**.

Syntax

```
unexpected_function _get_unexpected( void );
```

Return Value

Returns a pointer to the function registered by set_unexpected. If no function has been set, the return value may be used to restore the default behavior; this value may be **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_get_unexpected	<eh.h>

For additional compatibility information, see Compatibility.

See also

Exception Handling Routines
abort
set_terminate
terminate
unexpected

_get_wpgmptr

11/8/2018 • 2 minutes to read • Edit Online

Gets the current value of the **_wpgmptr** global variable.

Syntax

```
errno_t _get_wpgmptr(
   wchar_t **pValue
);
```

Parameters

pValue
A pointer to a string to be filled with the current value of the **_wpgmptr** variable.

Return Value

Returns zero if successful; an error code on failure. If *pValue* is **NULL**, the invalid parameter handler is invoked as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

Only call **_get_wpgmptr** if your program has a wide entry point, like **wmain()** or **wWinMain()**. The **_wpgmptr** global variable contains the full path to the executable associated with the process as a wide-character string. For more information, see _pgmptr, _wpgmptr.

Requirements

ROUTINE	REQUIRED HEADER
_get_wpgmptr	<stdlib.h>

For more compatibility information, see Compatibility.

See also

_get_pgmptr

getc, getwc

10/31/2018 • 2 minutes to read • Edit Online

Read a character from a stream.

Syntax

```
int getc(
   FILE *stream
);
wint_t getwc(
   FILE *stream
);
```

Parameters

stream
Input stream.

Return Value

Returns the character read. To indicate a read error or end-of-file condition, **getc** returns **EOF**, and **getwc** returns **WEOF**. For **getc**, use **ferror** or **feof** to check for an error or for end of file. If *stream* is **NULL**, **getc** and **getwc** invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** (or **WEOF** for **getwc**) and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

Each routine reads a single character from a file at the current position and increments the associated file pointer (if defined) to point to the next character. The file is associated with *stream*.

These functions lock the calling thread and are therefore thread-safe. For a non-locking version, see _getc_nolock, _getwc_nolock.

Routine-specific remarks follow.

ROUTINE	REMARKS
getc	Same as **fgetc**, but implemented as a function and as a macro.
getwc	Wide-character version of **getc**. Reads a multibyte character or a wide character according to whether *stream* is opened in text mode or binary mode.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettc	**getc**	**getc**	**getwc**

Requirements

ROUTINE	REQUIRED HEADER
getc	<stdio.h>
getwc	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_getc.c
// Use getc to read a line from a file.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;
    FILE* fp;

    // Read a single line from the file "crt_getc.txt".

    fopen_s(&fp, "crt_getc.txt", "r");
    if (!fp)
    {
       printf("Failed to open file crt_getc.txt.\n");
       exit(1);
    }

    for (i = 0; (i < 80) && ((ch = getc(fp)) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);

    fclose(fp);
}
```

Input: crt_getc.txt

```
Line one.
Line two.
```

Output

```
Input was: Line one.
```

See also

Stream I/O
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

_getc_nolock, _getwc_nolock

10/31/2018 • 2 minutes to read • Edit Online

Reads a character from a stream.

Syntax

```
int _getc_nolock(
   FILE *stream
);
wint_t _getwc_nolock(
   FILE *stream
);
```

Parameters

stream
Input stream.

Return Value

See getc, getwc.

Remarks

These functions are identical to **getc** and **getwc** except that they do not lock the calling thread. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettc_nolock	**getc_nolock**	**getc_nolock**	**getwc_nolock**

Requirements

ROUTINE	REQUIRED HEADER
getc_nolock	<stdio.h>
getwc_nolock	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getc_nolock.c
// Use getc to read a line from a file.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;
    FILE* fp;

    // Read a single line from the file "crt_getc_nolock.txt".
    fopen_s(&fp, "crt_getc_nolock.txt", "r");
    if (!fp)
    {
       printf("Failed to open file crt_getc_nolock.txt.\n");
       exit(1);
    }

    for (i = 0; (i < 80) && ((ch = getc(fp)) != EOF)
                         && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);

    fclose(fp);
}
```

Input: crt_getc_nolock.txt

```
Line the first.
Line the second.
```

Output

```
Input was: Line the first.
```

See also

Stream I/O
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

getch

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _getch instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_getch, _getwch

10/31/2018 • 2 minutes to read • Edit Online

Gets a character from the console without echo.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getch( void );
wint_t _getwch( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

The **_getch** and **_getwch** functions read a single character from the console without echoing the character. None of these functions can be used to read CTRL+C. When reading a function key or an arrow key, each function must be called twice; the first call returns 0 or 0xE0, and the second call returns the actual key code.

These functions lock the calling thread and are therefore thread-safe. For non-locking versions, see _getch_nolock, _getwch_nolock.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettch	**_getch**	**_getch**	**_getwch**

Requirements

ROUTINE	REQUIRED HEADER
_getch	<conio.h>
_getwch	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getch.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.

#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch( ch );
   _putch( '\r' );    // Carriage return
   _putch( '\n' );    // Line feed
}
```

```
abcdefy
```

```
Type 'Y' when finished typing keys: Y
```

See also

Console and Port I/O
_getche, _getwche
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

_getch_nolock, _getwch_nolock

10/31/2018 • 2 minutes to read • Edit Online

Gets a character from the console without echo and without locking the thread.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getch_nolock( void );
wint_t _getwch_nolock( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

_getch_nolock and **_getwch_nolock** are identical to **_getch** and **_getchw** except that they not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettch_nolock	**_getch_nolock**	**_getch_nolock**	**_getwch_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_getch_nolock	<conio.h>
_getwch_nolock	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getch_nolock.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.

#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch_nolock();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch_nolock( ch );
   _putch_nolock( '\r' );    // Carriage return
   _putch_nolock( '\n' );    // Line feed
}
```

```
abcdefy
```

```
Type 'Y' when finished typing keys: Y
```

See also

Console and Port I/O
_getche, _getwche
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

getchar, getwchar

10/31/2018 • 2 minutes to read • Edit Online

Reads a character from standard input.

Syntax

```
int getchar();
wint_t getwchar();
```

Return Value

Returns the character read. To indicate a read error or end-of-file condition, **getchar** returns **EOF**, and **getwchar** returns **WEOF**. For **getchar**, use **ferror** or **feof** to check for an error or for end of file.

Remarks

Each routine reads a single character from **stdin** and increments the associated file pointer to point to the next character. **getchar** is the same as _fgetchar, but it is implemented as a function and as a macro.

These functions lock the calling thread and are therefore thread-safe. For a non-locking version, see _getchar_nolock, _getwchar_nolock.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettchar	**getchar**	**getchar**	**getwchar**

Requirements

ROUTINE	REQUIRED HEADER
getchar	<stdio.h>
getwchar	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_getchar.c
// Use getchar to read a line from stdin.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;

    for (i = 0; (i < 80) && ((ch = getchar()) != EOF)
                        && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character
    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);
}
```

```
This textInput was: This text
```

See also

Stream I/O
getc, getwc
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

_getchar_nolock, _getwchar_nolock

10/31/2018 • 2 minutes to read • Edit Online

Reads a character from standard input.

Syntax

```
int _getchar_nolock( void );
wint_t _getwchar_nolock( void );
```

Return Value

See getchar, getwchar.

Remarks

_getchar_nolock and **_getwchar_nolock** are identical to **getchar** and **getwchar** except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettchar_nolock	**_getchar_nolock**	**_getchar_nolock**	**_getwchar_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_getchar_nolock	<stdio.h>
_getwchar_nolock	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_getchar_nolock.c
// Use _getchar_nolock to read a line from stdin.

#include <stdio.h>

int main()
{
    char buffer[81];
    int i, ch;

    for (i = 0; (i < 80) && ((ch = _getchar_nolock()) != EOF)
                        && (ch != '\n'); i++)
    {
        buffer[i] = (char) ch;
    }

    // Terminate string with a null character

    buffer[i] = '\0';
    printf( "Input was: %s\n", buffer);
}
```

```
This textInput was: This text
```

See also

Stream I/O
getc, getwc
fgetc, fgetwc
_getch, _getwch
putc, putwc
ungetc, ungetwc

getche

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _getche instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_getche, _getwche

10/31/2018 • 2 minutes to read • Edit Online

Gets a character from the console with echo.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getche( void );
wint_t _getwche( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

The **_getche** and **_getwche** functions read a single character from the console with echo, meaning that the character is displayed at the console. None of these functions can be used to read CTRL+C. When reading a function key or an arrow key, each function must be called twice; the first call returns 0 or 0xE0, and the second call returns the actual key code.

These functions lock the calling thread and are therefore thread-safe. For non-locking versions, see _getche_nolock, _getwche_nolock.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_getche	**_getche**	**_getch**	**_getwche**

Requirements

ROUTINE	REQUIRED HEADER
_getche	<conio.h>
_getwche	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getche.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.

#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getche();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch( ch );
   _putch( '\r' );    // Carriage return
   _putch( '\n' );    // Line feed
}
```

```
abcdefy
```

```
Type 'Y' when finished typing keys: abcdefyY
```

See also

Console and Port I/O
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

_getche_nolock, _getwche_nolock

10/31/2018 • 2 minutes to read • Edit Online

Gets a character from the console, with echo and without locking the thread.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getche_nolock( void );
wint_t _getwche_nolock( void );
```

Return Value

Returns the character read. There is no error return.

Remarks

_getche_nolock and **_getwche_nolock** are identical to **_getche** and **_getwche** except that they not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_gettche_nolock	**_getche_nolock**	**_getch_nolock**	**_getwche_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_getche_nolock	<conio.h>
_getwche_nolock	<conio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getche_nolock.c
// compile with: /c
// This program reads characters from
// the keyboard until it receives a 'Y' or 'y'.

#include <conio.h>
#include <ctype.h>

int main( void )
{
   int ch;

   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getche_nolock();
      ch = toupper( ch );
   } while( ch != 'Y' );

   _putch_nolock( ch );
   _putch_nolock( '\r' );    // Carriage return
   _putch_nolock( '\n' );    // Line feed
}
```

```
abcdefy
```

```
Type 'Y' when finished typing keys: abcdefyY
```

See also

Console and Port I/O
_cgets, _cgetws
getc, getwc
_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

getcwd

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _getcwd instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_getcwd, _wgetcwd

10/31/2018 • 2 minutes to read • Edit Online

Gets the current working directory.

Syntax

```
char *_getcwd(
   char *buffer,
   int maxlen
);
wchar_t *_wgetcwd(
   wchar_t *buffer,
   int maxlen
);
```

Parameters

buffer
Storage location for the path.

maxlen
Maximum length of the path in characters: **char** for **_getcwd** and **wchar_t** for **_wgetcwd**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters. If *maxlen* is less than or equal to zero, this function invokes an invalid parameter handler, as described in Parameter Validation.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_getcwd** function gets the full path of the current working directory for the default drive and stores it at *buffer*. The integer argument *maxlen* specifies the maximum length for the path. An error occurs if the length of the path (including the terminating null character) exceeds *maxlen*. The *buffer* argument can be **NULL**; a buffer of at least size *maxlen* (more only if necessary) is automatically allocated, using **malloc**, to store the path. This buffer can later be freed by calling **free** and passing it the **_getcwd** return value (a pointer to the allocated buffer).

_getcwd returns a string that represents the path of the current working directory. If the current working directory is the root, the string ends with a backslash (****). If the current working directory is a directory other than the root, the string ends with the directory name and not with a backslash.

_wgetcwd is a wide-character version of **_getcwd**; the *buffer* argument and return value of **_wgetcwd** are wide-character strings. **_wgetcwd** and **_getcwd** behave identically otherwise.

When **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, calls to **_getcwd** and **_wgetcwd** are replaced by calls to **_getcwd_dbg** and **_wgetcwd_dbg** to allow for debugging memory allocations. For more information, see _getcwd_dbg, _wgetcwd_dbg.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetcwd	**_getcwd**	**_getcwd**	**_wgetcwd**

Requirements

ROUTINE	REQUIRED HEADER
_getcwd	<direct.h>
_wgetcwd	<direct.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getcwd.c
// This program places the name of the current directory in the
// buffer array, then displays the name of the current directory
// on the screen. Passing NULL as the buffer forces getcwd to allocate
// memory for the path, which allows the code to support file paths
// longer than _MAX_PATH, which are supported by NTFS.

#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char* buffer;

   // Get the current working directory:
   if( (buffer = _getcwd( NULL, 0 )) == NULL )
      perror( "_getcwd error" );
   else
   {
      printf( "%s \nLength: %d\n", buffer, strnlen(buffer) );
      free(buffer);
   }
}
```

```
C:\Code
```

See also

Directory Control
_chdir, _wchdir
_mkdir, _wmkdir
_rmdir, _wrmdir

_getcwd_dbg, _wgetcwd_dbg

10/31/2018 • 2 minutes to read • Edit Online

Debug versions of the _getcwd, _wgetcwd functions (only available during debug).

Syntax

```
char *_getcwd_dbg(
   char *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wgetcwd_dbg(
   wchar_t *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

buffer
Storage location for the path.

maxlen
Maximum length of the path in characters: **char** for **_getcwd_dbg** and **wchar_t** for **_wgetcwd_dbg**.

blockType
Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber
Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters.

For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_getcwd_dbg** and **_wgetcwd_dbg** functions are identical to **_getcwd** and **_wgetcwd** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg** to allocate memory if **NULL** is passed as the first parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_getcwd** and **_wgetcwd** are

remapped to **_getcwd_dbg** and **_wgetcwd_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of blocks on the debug heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetcwd_dbg	**_getcwd_dbg**	**_getcwd_dbg**	**_wgetcwd_dbg**

Requirements

ROUTINE	REQUIRED HEADER
_getcwd_dbg	<crtdbg.h>
_wgetcwd_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

See also

_getcwd, _wgetcwd
Directory Control
Debug Versions of Heap Allocation Functions

_getdcwd, _wgetdcwd

2/4/2019 • 2 minutes to read • Edit Online

Gets the full path of the current working directory on the specified drive.

Syntax

```
char *_getdcwd(
   int drive,
   char *buffer,
   int maxlen
);
wchar_t *_wgetdcwd(
   int drive,
   wchar_t *buffer,
   int maxlen
);
```

Parameters

drive
A non-negative integer that specifies the drive (0 = default drive, 1 = A, 2 = B, and so on).

If the specified drive isn't available, or the kind of drive (for example, removable, fixed, CD-ROM, RAM disk, or network drive) can't be determined, the invalid-parameter handler is invoked. For more information, see Parameter Validation.

buffer
Storage location for the path, or **NULL**.

If **NULL** is specified, this function allocates a buffer of at least *maxlen* size by using **malloc**, and the return value of **_getdcwd** is a pointer to the allocated buffer. The buffer can be freed by calling **free** and passing it the pointer.

maxlen
A nonzero positive integer that specifies the maximum length of the path, in characters: **char** for **_getdcwd** and **wchar_t** for **_wgetdcwd**.

If *maxlen* is less than or equal to zero, the invalid-parameter handler is invoked. For more information, see Parameter Validation.

Return Value

Pointer to a string that represents the full path of the current working directory on the specified drive, or **NULL**, which indicates an error.

If *buffer* is specified as **NULL** and there is insufficient memory to allocate *maxlen* characters, an error occurs and **errno** is set to **ENOMEM**. If the length of the path including the terminating null character exceeds *maxlen*, an error occurs, and **errno** is set to **ERANGE**. For more information about these error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_getdcwd** function gets the full path of the current working directory on the specified drive and stores it at *buffer*. If the current working directory is set to the root, the string ends with a backslash (\). If the current

working directory is set to a directory other than the root, the string ends with the name of the directory and not with a backslash.

_wgetdcwd is a wide-character version of **_getdcwd**, and its *buffer* parameter and return value are wide-character strings. Otherwise, **_wgetdcwd** and **_getdcwd** behave identically.

This function is thread-safe even though it depends on **GetFullPathName**, which is itself not thread-safe. However, you can violate thread safety if your multithreaded application calls both this function and GetFullPathNameA.

The version of this function that has the **_nolock** suffix behaves identically to this function except that it is not thread-safe and is not protected from interference by other threads. For more information, see _getdcwd_nolock, _wgetdcwd_nolock.

When **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, calls to **_getdcwd** and **_wgetdcwd** are replaced by calls to **_getdcwd_dbg** and **_wgetdcwd_dbg** so that you can debug memory allocations. For more information, see_getdcwd_dbg, _wgetdcwd_dbg.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetdcwd	**_getdcwd**	**_getdcwd**	**_wgetdcwd**

Requirements

ROUTINE	REQUIRED HEADER
_getdcwd	<direct.h>
_wgetdcwd	<direct.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _getdrive.

See also

Directory Control
_chdir, _wchdir
_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir

_getdcwd_dbg, _wgetdcwd_dbg

10/31/2018 • 2 minutes to read • Edit Online

Debug versions of the _getdcwd, _wgetdcwd functions (only available during debug).

Syntax

```
char *_getdcwd_dbg(
   int drive,
   char *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wgetdcwd_dbg(
   int drive,
   wchar_t *buffer,
   int maxlen,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

drive
Name of the disk drive.

buffer
Storage location for the path.

maxlen
Maximum length of the path in characters: **char** for **_getdcwd_dbg** and **wchar_t** for **_wgetdcwd_dbg**.

blockType
Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber
Line number in the source file where the allocation operation was requested or **NULL**.

Return Value

Returns a pointer to *buffer*. A **NULL** return value indicates an error, and **errno** is set either to **ENOMEM**, indicating that there is insufficient memory to allocate *maxlen* bytes (when a **NULL** argument is given as *buffer*), or to **ERANGE**, indicating that the path is longer than *maxlen* characters. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_getdcwd_dbg** and **_wgetdcwd_dbg** functions are identical to **_getdcwd** and **_wgetdcwd** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg** to allocate memory

if **NULL** is passed as the *buffer* parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the **_CRTDBG_MAP_ALLOC** flag. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_getdcwd** and **_wgetdcwd** are remapped to **_getdcwd_dbg** and **_wgetdcwd_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of Blocks on the Debug Heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetdcwd_dbg	**_getdcwd_dbg**	**_getdcwd_dbg**	**_wgetdcwd_dbg**

Requirements

ROUTINE	REQUIRED HEADER
_getdcwd_dbg	<crtdbg.h>
_wgetdcwd_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

See also

_getdcwd, _wgetdcwd
Directory Control
Debug Versions of Heap Allocation Functions

_getdcwd_nolock, _wgetdcwd_nolock

10/31/2018 • 2 minutes to read • Edit Online

Gets the full path of the current working directory on the specified drive.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_getdcwd_nolock(
   int drive,
   char *buffer,
   int maxlen
);
wchar_t *_wgetdcwd_nolock(
   int drive,
   wchar_t *buffer,
   int maxlen
);
```

Parameters

drive
Disk drive.

buffer
Storage location for the path.

maxlen
Maximum length of path in characters: **char** for **_getdcwd** and **wchar_t** for **_wgetdcwd**.

Return Value

See _getdcwd, _wgetdcwd.

Remarks

_getdcwd_nolock and **_wgetdcwd_nolock** are identical to **_getdcwd** and **_wgetdcwd**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetdcwd_nolock	**_getdcwd_nolock**	**_getdcwd_nolock**	**_wgetdcwd_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_getdcwd_nolock	<direct.h>
_wgetdcwd_nolock	<direct.h> or <wchar.h>

For more compatibility information, see Compatibility.

See also

Directory Control
_chdir, _wchdir
_getcwd, _wgetcwd
_getdrive
_mkdir, _wmkdir
_rmdir, _wrmdir

_getdiskfree

10/31/2018 • 2 minutes to read • Edit Online

Uses information about a disk drive to populate a **_diskfree_t** structure.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned _getdiskfree(
   unsigned drive,
   struct _diskfree_t * driveinfo
);
```

Parameters

drive
The disk drive for which you want information.

driveinfo
A **_diskfree_t** structure that will be populated with information about the drive.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is the error code. The value **errno** is set for any errors that are returned by the operating system. For more information about error conditions that are indicated by **errno**, see errno Constants.

Remarks

The **_diskfree_t** structure is defined in Direct.h.

```
struct _diskfree_t {
   unsigned total_clusters;      // The total number of clusters, both used and available, on the disk.
   unsigned avail_clusters;      // The number of unused clusters on the disk.
   unsigned sectors_per_cluster; // The number of sectors in each cluster.
   unsigned bytes_per_sector;    // The size of each sector in bytes.
};
```

This function validates its parameters. If the *driveinfo* pointer is **NULL** or *drive* specifies an invalid drive, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**. Valid drives range from 0 to 26. A *drive* value of 0 specifies the current drive; thereafter, numbers map to letters of the English alphabet such that 1 indicates drive A, 3 indicates drive C, and so on.

Requirements

ROUTINE	REQUIRED HEADER
_getdiskfree	<direct.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getdiskfree.c
// compile with: /c
#include <windows.h>
#include <direct.h>
#include <stdio.h>
#include <tchar.h>

TCHAR   g_szBorder[] = _T("=========================================================================\n");
TCHAR   g_szTitle1[] = _T("|DRIVE|TOTAL CLUSTERS|AVAIL CLUSTERS|SECTORS / CLUSTER|BYTES / SECTOR|\n");
TCHAR   g_szTitle2[] = _T("|=====|==============|==============|=================|==============|\n");
TCHAR   g_szLine[]   = _T("|  A: |              |              |                 |              |\n");

void utoiRightJustified(TCHAR* szLeft, TCHAR* szRight, unsigned uVal);

int main(int argc, char* argv[]) {
   TCHAR szMsg[4200];
   struct _diskfree_t df = {0};
   ULONG uDriveMask = _getdrives();
   unsigned uErr, uLen, uDrive;

   printf(g_szBorder);
   printf(g_szTitle1);
   printf(g_szTitle2);

   for (uDrive=1; uDrive<=26; ++uDrive) {
      if (uDriveMask & 1) {
         uErr = _getdiskfree(uDrive, &df);
         memcpy(szMsg, g_szLine, sizeof(g_szLine));
         szMsg[3] = uDrive + 'A' - 1;

         if (uErr == 0) {
            utoiRightJustified(szMsg+8,  szMsg+19, df.total_clusters);
            utoiRightJustified(szMsg+23, szMsg+34, df.avail_clusters);
            utoiRightJustified(szMsg+38, szMsg+52, df.sectors_per_cluster);
            utoiRightJustified(szMsg+56, szMsg+67, df.bytes_per_sector);
         }
         else {
            uLen = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
                           uErr, 0, szMsg+8, 4100, NULL);
            szMsg[uLen+6] = ' ';
            szMsg[uLen+7] = ' ';
            szMsg[uLen+8] = ' ';
         }

         printf(szMsg);
      }

      uDriveMask >>= 1;
   }

   printf(g_szBorder);
}

void utoiRightJustified(TCHAR* szLeft, TCHAR* szRight, unsigned uVal) {
   TCHAR* szCur = szRight;
   int nComma = 0;
```

```
   if (uVal) {
      while (uVal && (szCur >= szLeft)) {
         if   (nComma == 3) {
            *szCur = ',';
            nComma = 0;
         }
         else {
            *szCur = (uVal % 10) | 0x30;
            uVal /= 10;
            ++nComma;
         }

         --szCur;
      }
   }
   else {
      *szCur = '0';
      --szCur;
   }

   if (uVal) {
      szCur = szLeft;

      while   (szCur <= szRight) {
         *szCur = '*';
         ++szCur;
      }
   }
}
```

```
=====================================================================
|DRIVE|TOTAL CLUSTERS|AVAIL CLUSTERS|SECTORS / CLUSTER|BYTES / SECTOR|
|=====|==============|==============|=================|==============|
|  A: | The device is not ready.    |                 |              |
|  C: |    4,721,093 |    3,778,303 |               8 |          512 |
|  D: |    1,956,097 |    1,800,761 |               8 |          512 |
|  E: | The device is not ready.    |                 |              |
=====================================================================
```

See also

Directory Control

_getdrive

10/31/2018 • 2 minutes to read • Edit Online

Gets the current disk drive.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getdrive( void );
```

Return Value

Returns the current (default) drive (1=A, 2=B, and so on). There is no error return.

Requirements

ROUTINE	REQUIRED HEADER
_getdrive	<direct.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getdrive.c
// compile with: /c
// Illustrates drive functions including:
//    _getdrive       _chdrive        _getdcwd
//

#include <stdio.h>
#include <direct.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];

   // Save current drive.
   curdrive = _getdrive();

   printf( "Available drives are:\n" );

   // If we can switch to the drive, it exists.
   for( drive = 1; drive <= 26; drive++ )
   {
      if( !_chdrive( drive ) )
      {
         printf( "%c:", drive + 'A' - 1 );
         if( _getdcwd( drive, path, _MAX_PATH ) != NULL )
            printf( " (Current directory is %s)", path );
         putchar( '\n' );
      }
   }

   // Restore original drive.
   _chdrive( curdrive );
}
```

```
Available drives are:
A: (Current directory is A:\)
C: (Current directory is C:\)
E: (Current directory is E:\testdir\bin)
F: (Current directory is F:\)
G: (Current directory is G:\)
```

See also

Directory Control
_chdrive
_getcwd, _wgetcwd
_getdcwd, _wgetdcwd

_getdrives

11/14/2018 • 2 minutes to read • Edit Online

Returns a bitmask that represents the currently available disk drives.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned long _getdrives( void );
```

Return Value

If the function succeeds, the return value is a bitmask that represents the currently available disk drives. Bit position 0 (the least-significant bit) is drive A, bit position 1 is drive B, bit position 2 is drive C, and so on. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

ROUTINE	REQUIRED HEADER
_getdrives	<direct.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getdrives.c
// This program retrieves and lists out
// all the logical drives that are
// currently mounted on the machine.

#include <windows.h>
#include <direct.h>
#include <stdio.h>
#include <tchar.h>

TCHAR g_szDrvMsg[] = _T("A:\n");

int main(int argc, char* argv[]) {
   ULONG uDriveMask = _getdrives();

   if (uDriveMask == 0)
   {
      printf( "_getdrives() failed with failure code: %d\n",
              GetLastError());
   }
   else
   {
      printf("The following logical drives are being used:\n");

      while (uDriveMask) {
         if (uDriveMask & 1)
            printf(g_szDrvMsg);

         ++g_szDrvMsg[0];
         uDriveMask >>= 1;
      }
   }
}
```

```
The following logical drives are being used:
A:
C:
D:
E:
```

See also

Directory Control

getenv, _wgetenv

10/31/2018 • 3 minutes to read • Edit Online

Gets a value from the current environment. More secure versions of these functions are available; see getenv_s, _wgetenv_s.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *getenv(
   const char *varname
);
wchar_t *_wgetenv(
   const wchar_t *varname
);
```

Parameters

varname
Environment variable name.

Return Value

Returns a pointer to the environment table entry containing *varname*. It is not safe to modify the value of the environment variable using the returned pointer. Use the _putenv function to modify the value of an environment variable. The return value is **NULL** if *varname* is not found in the environment table.

Remarks

The **getenv** function searches the list of environment variables for *varname*. **getenv** is not case sensitive in the Windows operating system. **getenv** and **_putenv** use the copy of the environment pointed to by the global variable **_environ** to access the environment. **getenv** operates only on the data structures accessible to the run-time library and not on the environment "segment" created for the process by the operating system. Therefore, programs that use the *envp* argument to main or wmain may retrieve invalid information.

If *varname* is **NULL**, this function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **NULL**.

_wgetenv is a wide-character version of **getenv**; the argument and return value of **_wgetenv** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**.

In an MBCS program (for example, in an SBCS ASCII program), **_wenviron** is initially **NULL** because the environment is composed of multibyte-character strings. Then, on the first call to _wputenv, or on the first call to **_wgetenv** if an (MBCS) environment already exists, a corresponding wide-character string environment is created and is then pointed to by **_wenviron**.

Similarly in a Unicode (**_wmain**) program, **_environ** is initially **NULL** because the environment is composed of

wide-character strings. Then, on the first call to **_putenv**, or on the first call to **getenv** if a (Unicode) environment already exists, a corresponding MBCS environment is created and is then pointed to by **_environ**.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, resulting in slower execution time. For example, whenever you call **_putenv**, a call to **_wputenv** is also executed automatically, so that the two environment strings correspond.

Caution

In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, these two environment versions may not correspond exactly. This is because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. For more information, see _environ, _wenviron.

> **NOTE**
> The **_putenv** and **_getenv** families of functions are not thread-safe. **_getenv** could return a string pointer while **_putenv** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetenv	**getenv**	**getenv**	**_wgetenv**

To check or change the value of the **TZ** environment variable, use **getenv**, **_putenv** and **_tzset** as necessary. For more information about **TZ**, see _tzset and _daylight, timezone, and _tzname.

Requirements

ROUTINE	REQUIRED HEADER
getenv	<stdlib.h>
_wgetenv	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_getenv.c
// compile with: /W3
// This program uses getenv to retrieve
// the LIB environment variable and then uses
// _putenv to change it to a new value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *libvar;

   // Get the value of the LIB environment variable.
   libvar = getenv( "LIB" ); // C4996
   // Note: getenv is deprecated; consider using getenv_s instead

   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );

   // Attempt to change path. Note that this only affects the environment
   // variable of the current process. The command processor's
   // environment is not changed.
   _putenv( "LIB=c:\\mylib;c:\\yourlib" ); // C4996
   // Note: _putenv is deprecated; consider using putenv_s instead

   // Get new value.
   libvar = getenv( "LIB" ); // C4996

   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );
}
```

```
Original LIB variable is: C:\progra~1\devstu~1\vc\lib
New LIB variable is: c:\mylib;c:\yourlib
```

See also

Process and Environment Control
_putenv, _wputenv
Environmental Constants

getenv_s, _wgetenv_s

10/31/2018 • 4 minutes to read • Edit Online

Gets a value from the current environment. These versions of getenv, _wgetenv have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t getenv_s(
   size_t *pReturnValue,
   char* buffer,
   size_t numberOfElements,
   const char *varname
);
errno_t _wgetenv_s(
   size_t *pReturnValue,
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *varname
);
template <size_t size>
errno_t getenv_s(
   size_t *pReturnValue,
   char (&buffer)[size],
   const char *varname
); // C++ only
template <size_t size>
errno_t _wgetenv_s(
   size_t *pReturnValue,
   wchar_t (&buffer)[size],
   const wchar_t *varname
); // C++ only
```

Parameters

pReturnValue
The buffer size that's required, or 0 if the variable is not found.

buffer
Buffer to store the value of the environment variable.

numberOfElements
Size of *buffer*.

varname
Environment variable name.

Return Value

Zero if successful; otherwise, an error code on failure.

Error Conditions

PRETURNVALUE	*BUFFER*	*NUMBEROFELEMENTS*	*VARNAME*	RETURN VALUE
NULL	any	any	any	**EINVAL**
any	**NULL**	>0	any	**EINVAL**
any	any	any	**NULL**	**EINVAL**

Any of these error conditions invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EINVAL**.

Also, if the buffer is too small, these functions return **ERANGE**. They do not invoke an invalid parameter handler. They write out the required buffer size in *pReturnValue*, and thereby enable programs to call the function again with a larger buffer.

Remarks

The **getenv_s** function searches the list of environment variables for *varname*. **getenv_s** is not case sensitive in the Windows operating system. **getenv_s** and _putenv_s use the copy of the environment that's pointed to by the global variable **_environ** to access the environment. **getenv_s** operates only on the data structures that are accessible to the run-time library and not on the environment "segment" that's created for the process by the operating system. Therefore, programs that use the *envp* argument to main or wmain might retrieve invalid information.

_wgetenv_s is a wide-character version of **getenv_s**; the argument and return value of **_wgetenv_s** are wide-character strings. The **_wenviron** global variable is a wide-character version of **_environ**.

In an MBCS program (for example, in an SBCS ASCII program), **_wenviron** is initially **NULL** because the environment is composed of multibyte-character strings. Then, on the first call to _wputenv, or on the first call to **_wgetenv_s**, if an (MBCS) environment already exists, a corresponding wide-character string environment is created and is then pointed to by **_wenviron**.

Similarly in a Unicode (**_wmain**) program, **_environ** is initially **NULL** because the environment is composed of wide-character strings. Then, on the first call to _putenv, or on the first call to **getenv_s** if a (Unicode) environment already exists, a corresponding MBCS environment is created and is then pointed to by **_environ**.

When two copies of the environment (MBCS and Unicode) exist simultaneously in a program, the run-time system must maintain both copies, and this causes slower execution time. For example, when you call **_putenv**, a call to **_wputenv** is also executed automatically so that the two environment strings correspond.

Caution

In rare instances, when the run-time system is maintaining both a Unicode version and a multibyte version of the environment, the two environment versions may not correspond exactly. This happens because, although any unique multibyte-character string maps to a unique Unicode string, the mapping from a unique Unicode string to a multibyte-character string is not necessarily unique. For more information, see _environ, _wenviron.

> **NOTE**
>
> The **_putenv_s** and **_getenv_s** families of functions are not thread-safe. **_getenv_s** could return a string pointer while **_putenv_s** is modifying the string and thereby cause random failures. Make sure that calls to these functions are synchronized.

In C++, use of these functions is simplified by template overloads; the overloads can infer buffer length automatically and thereby eliminate the need to specify a size argument. For more information, see Secure

Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tgetenv_s	**getenv_s**	**getenv_s**	**_wgetenv_s**

To check or change the value of the **TZ** environment variable, use **getenv_s**, **_putenv**, and **_tzset**, as required. For more information about **TZ**, see _tzset and _daylight, _dstbias, _timezone, and _tzname.

Requirements

ROUTINE	REQUIRED HEADER
getenv_s	<stdlib.h>
_wgetenv_s	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_getenv_s.c
// This program uses getenv_s to retrieve
// the LIB environment variable and then uses
// _putenv to change it to a new value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char* libvar;
   size_t requiredSize;

   getenv_s( &requiredSize, NULL, 0, "LIB");
   if (requiredSize == 0)
   {
      printf("LIB doesn't exist!\n");
      exit(1);
   }

   libvar = (char*) malloc(requiredSize * sizeof(char));
   if (!libvar)
   {
      printf("Failed to allocate memory!\n");
      exit(1);
   }

   // Get the value of the LIB environment variable.
   getenv_s( &requiredSize, libvar, requiredSize, "LIB" );

   printf( "Original LIB variable is: %s\n", libvar );

   // Attempt to change path. Note that this only affects
   // the environment variable of the current process. The command
   // processor's environment is not changed.
   _putenv_s( "LIB", "c:\\mylib;c:\\yourlib" );

   getenv_s( &requiredSize, NULL, 0, "LIB");

   libvar = (char*) realloc(libvar, requiredSize * sizeof(char));
   if (!libvar)
   {
      printf("Failed to allocate memory!\n");
      exit(1);
   }

   // Get the new value of the LIB environment variable.
   getenv_s( &requiredSize, libvar, requiredSize, "LIB" );

   printf( "New LIB variable is: %s\n", libvar );

   free(libvar);
}
```

```
Original LIB variable is: c:\vctools\lib;c:\vctools\atlmfc\lib;c:\vctools\PlatformSDK\lib;c:\vctools\Visual
Studio SDKs\DIA Sdk\lib;c:\vctools\Visual Studio SDKs\BSC Sdk\lib
New LIB variable is: c:\mylib;c:\yourlib
```

See also

Process and Environment Control
Environmental Constants
_putenv, _wputenv

_dupenv_s, _wdupenv_s

_getmaxstdio

10/31/2018 • 2 minutes to read • Edit Online

Returns the number of simultaneously open files permitted at the stream I/O level.

Syntax

```
int _getmaxstdio( void );
```

Return Value

Returns a number that represents the number of simultaneously open files currently permitted at the **stdio** level.

Remarks

Use _setmaxstdio to configure the number of simultaneously open files permitted at the **stdio** level.

Requirements

ROUTINE	REQUIRED HEADER
_getmaxstdio	<stdio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_setmaxstdio.c
// The program retrieves the maximum number
// of open files and prints the results
// to the console.

#include <stdio.h>

int main()
{
   printf( "%d\n", _getmaxstdio());

   _setmaxstdio(2048);

   printf( "%d\n", _getmaxstdio());
}
```

```
512
2048
```

See also

Stream I/O

_getmbcp

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the current code page.

Syntax

```
int _getmbcp( void );
```

Return Value

Returns the current multibyte code page. A return value of 0 indicates that a single byte code page is in use.

Requirements

ROUTINE	REQUIRED HEADER
_getmbcp	<mbctype.h>

For more compatibility information, see Compatibility.

See also

_setmbcp

getpid

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _getpid instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_getpid

10/31/2018 • 2 minutes to read • Edit Online

Gets the process identification.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _getpid( void );
```

Return Value

Returns the process ID obtained from the system. There is no error return.

Remarks

The **_getpid** function obtains the process ID from the system. The process ID uniquely identifies the calling process.

Requirements

ROUTINE	REQUIRED HEADER
_getpid	<process.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getpid.c
// This program uses _getpid to obtain
// the process ID and then prints the ID.

#include <stdio.h>
#include <process.h>

int main( void )
{
   // If run from command line, shows different ID for
   // command line than for operating system shell.

   printf( "Process id: %d\n", _getpid() );
}
```

```
Process id: 3584
```

See also

Process and Environment Control
_mktemp, _wmktemp

gets_s, _getws_s

10/31/2018 • 2 minutes to read • Edit Online

Gets a line from the **stdin** stream. These versions of gets, _getws have security enhancements, as described in Security Features in the CRT.

Syntax

```
char *gets_s(
   char *buffer,
   size_t sizeInCharacters
);
wchar_t *_getws_s(
   wchar_t *buffer,
   size_t sizeInCharacters
);
```

```
template <size_t size>
char *gets_s( char (&buffer)[size] ); // C++ only

template <size_t size>
wchar_t *_getws_s( wchar_t (&buffer)[size] ); // C++ only
```

Parameters

buffer
Storage location for input string.

sizeInCharacters
The size of the buffer.

Return Value

Returns *buffer* if successful. A **NULL** pointer indicates an error or end-of-file condition. Use ferror or feof to determine which one has occurred.

Remarks

The **gets_s** function reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first newline character ('\n'). **gets_s** then replaces the newline character with a null character ('\0') before returning the line. In contrast, the **fgets_s** function retains the newline character.

If the first character read is the end-of-file character, a null character is stored at the beginning of *buffer* and **NULL** is returned.

_getws_s is a wide-character version of **gets_s**; its argument and return value are wide-character strings.

If *buffer* is **NULL** or *sizeInCharacters* is less than or equal to zero, or if the buffer is too small to contain the input line and null terminator, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set errno to **ERANGE**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-

secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_getts_s	**gets_s**	**gets_s**	**_getws_s**

Requirements

ROUTINE	REQUIRED HEADER
gets_s	<stdio.h>
_getws_s	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_gets_s.c
// This program retrieves a string from the stdin and
// prints the same string to the console.

#include <stdio.h>

int main( void )
{
   char line[21]; // room for 20 chars + '\0'
   gets_s( line, 20 );
   printf( "The line entered was: %s\n", line );
}
```

```
Hello there!
```

```
The line entered was: Hello there!
```

See also

Stream I/O
gets, _getws
fgets, fgetws
fputs, fputws
puts, _putws

getw

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _getw instead.

_getw

10/31/2018 • 2 minutes to read • Edit Online

Gets an integer from a stream.

Syntax

```
int _getw(
   FILE *stream
);
```

Parameters

stream
Pointer to the **FILE** structure.

Return Value

_getw returns the integer value read. A return value of **EOF** indicates either an error or end of file. However, because the **EOF** value is also a legitimate integer value, use **feof** or **ferror** to verify an end-of-file or error condition. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EOF**.

Remarks

The **_getw** function reads the next binary value of type **int** from the file associated with *stream* and increments the associated file pointer (if there is one) to point to the next unread character. **_getw** does not assume any special alignment of items in the stream. Problems with porting can occur with **_getw** because the size of the **int** type and the ordering of bytes within the **int** type differ across systems.

Requirements

ROUTINE	REQUIRED HEADER
_getw	<stdio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_getw.c
// This program uses _getw to read a word
// from a stream, then performs an error check.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   int i;

   if( fopen_s( &stream, "crt_getw.txt", "rb" ) )
      printf( "Couldn't open file\n" );
   else
   {
      // Read a word from the stream:
      i = _getw( stream );

      // If there is an error...
      if( ferror( stream ) )
      {
         printf( "_getw failed\n" );
         clearerr_s( stream );
      }
      else
         printf( "First data word in file: 0x%.4x\n", i );
      fclose( stream );
   }
}
```

Input: crt_getw.txt

```
Line one.
Line two.
```

Output

```
First data word in file: 0x656e694c
```

See also

Stream I/O
_putw

gmtime, _gmtime32, _gmtime64

10/31/2018 • 3 minutes to read • Edit Online

Converts a **time_t** time value to a **tm** structure. More secure versions of these functions are available; see gmtime_s, _gmtime32_s, _gmtime64_s.

Syntax

```
struct tm *gmtime( const time_t *sourceTime );
struct tm *_gmtime32( const __time32_t *sourceTime );
struct tm *_gmtime64( const __time64_t *sourceTime );
```

Parameters

sourceTime
Pointer to the stored time. The time is represented as seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

Return Value

A pointer to a structure of type tm. The fields of the returned structure hold the evaluated value of the *sourceTime* argument in UTC rather than in local time. Each of the structure fields is of type **int**, as follows:

FIELD	DESCRIPTION
tm_sec	Seconds after minute (0 - 59).
tm_min	Minutes after hour (0 - 59).
tm_hour	Hours since midnight (0 - 23).
tm_mday	Day of month (1 - 31).
tm_mon	Month (0 - 11; January = 0).
tm_year	Year (current year minus 1900).
tm_wday	Day of week (0 - 6; Sunday = 0).
tm_yday	Day of year (0 - 365; January 1 = 0).
tm_isdst	Always 0 for **gmtime**.

Both the 32-bit and 64-bit versions of **gmtime**, mktime, mkgmtime, and localtime all use one common **tm** structure per thread for the conversion. Each call to one of these functions destroys the result of any previous call. If *sourceTime* represents a date before midnight, January 1, 1970, **gmtime** returns **NULL**. There is no error return.

_gmtime64, which uses the **__time64_t** structure, enables dates to be expressed up through 23:59:59, December 31, 3000, UTC, whereas **_gmtime32** only represent dates through 23:59:59 January 18, 2038,

UTC. Midnight, January 1, 1970, is the lower bound of the date range for both functions.

gmtime is an inline function that evaluates to **_gmtime64**, and **time_t** is equivalent to **__time64_t** unless **_USE_32BIT_TIME_T** is defined. If you must force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**, but doing so causes **gmtime** to be in-lined to **_gmtime32** and **time_t** to be defined as **__time32_t**. We recommend that you do not do this, because it is not allowed on 64-bit platforms and in any case your application may fail after January 18, 2038.

These functions validate their parameters. If *sourceTime* is a null pointer, or if the *sourceTime* value is negative, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set **errno** to **EINVAL**.

Remarks

The **_gmtime32** function breaks down the *sourceTime* value and stores it in a statically allocated structure of type **tm**, defined in TIME.H. The value of *sourceTime* is typically obtained from a call to the time function.

> **NOTE**
> In most cases, the target environment tries to determine whether daylight savings time is in effect. The C run-time library assumes that the United States rules for implementing the calculation of Daylight Saving Time (DST) are used.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
gmtime, **_gmtime32**, **_gmtime64**	<time.h>	<ctime> or <time.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_gmtime.c
// compile with: /W3
// This program uses _gmtime64 to convert a long-
// integer representation of coordinated universal time
// to a structure named newtime, then uses asctime to
// convert this structure to an output string.

#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm *newtime;
   __int64 ltime;
   char buff[80];

   _time64( &ltime );

   // Obtain coordinated universal time:
   newtime = _gmtime64( &ltime ); // C4996
   // Note: _gmtime64 is deprecated; consider using _gmtime64_s
   asctime_s( buff, sizeof(buff), newtime );
   printf( "Coordinated universal time is %s\n", buff );
}
```

```
Coordinated universal time is Tue Feb 12 23:11:31 2002
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
_mkgmtime, _mkgmtime32, _mkgmtime64
mktime, _mktime32, _mktime64
time, _time32, _time64

gmtime_s, _gmtime32_s, _gmtime64_s

11/8/2018 • 3 minutes to read • Edit Online

Converts a time value to a **tm** structure. These are versions of _gmtime32, _gmtime64 with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t gmtime_s(
   struct tm* tmDest,
   const __time_t* sourceTime
);
errno_t _gmtime32_s(
   struct tm* tmDest,
   const __time32_t* sourceTime
);
errno_t _gmtime64_s(
   struct tm* tmDest,
   const __time64_t* sourceTime
);
```

Parameters

tmDest
Pointer to a tm structure. The fields of the returned structure hold the evaluated value of the *timer* argument in UTC rather than in local time.

sourceTime
Pointer to stored time. The time is represented as seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC).

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h; for a listing of these errors, see errno.

Error Conditions

TMDEST	*SOURCETIME*	RETURN	VALUE IN *TMDEST*
NULL	any	**EINVAL**	Not modified.
Not **NULL** (points to valid memory)	**NULL**	**EINVAL**	All fields set to -1.
Not **NULL**	< 0	**EINVAL**	All fields set to -1.

In the case of the first two error conditions, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

The **_gmtime32_s** function breaks down the *sourceTime* value and stores it in a structure of type **tm**, defined in

Time.h. The address of the structure is passed in *tmDest*. The value of *sourceTime* is usually obtained from a call to the time function.

> **NOTE**
> The target environment should try to determine whether daylight savings time is in effect. The C run-time library assumes the United States rules for implementing the calculation of daylight saving time .

Each of the structure fields is of type **int**, as shown in the following table.

FIELD	DESCRIPTION
tm_sec	Seconds after minute (0 - 59).
tm_min	Minutes after hour (0 - 59).
tm_hour	Hours since midnight (0 - 23).
tm_mday	Day of month (1 - 31).
tm_mon	Month (0 - 11; January = 0).
tm_year	Year (current year minus 1900).
tm_wday	Day of week (0 - 6; Sunday = 0).
tm_yday	Day of year (0 - 365; January 1 = 0).
tm_isdst	Always 0 for **gmtime_s**.

_gmtime64_s, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, UTC; whereas **gmtime32_s** only represent dates through 23:59:59 January 18, 2038, UTC. Midnight, January 1, 1970, is the lower bound of the date range for both these functions.

gmtime_s is an inline function which evaluates to **_gmtime64_s** and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **gmtime_s** to be in-lined to **_gmtime32_s**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
gmtime_s, **_gmtime32_s**, **_gmtime64_s**	<time.h>	<ctime> or <time.h>

For more compatibility information, see Compatibility.

Example

```
// crt_gmtime64_s.c
// This program uses _gmtime64_s to convert a 64-bit
// integer representation of coordinated universal time
// to a structure named newtime, then uses asctime_s to
// convert this structure to an output string.

#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm newtime;
   __int64 ltime;
   char buf[26];
   errno_t err;

   _time64( &ltime );

   // Obtain coordinated universal time:
   err = _gmtime64_s( &newtime, &ltime );
   if (err)
   {
      printf("Invalid Argument to _gmtime64_s.");
   }

   // Convert to an ASCII representation
   err = asctime_s(buf, 26, &newtime);
   if (err)
   {
      printf("Invalid Argument to asctime_s.");
   }

   printf( "Coordinated universal time is %s\n",
           buf );
}
```

```
Coordinated universal time is Fri Apr 25 20:12:33 2003
```

See also

Time Management
asctime_s, _wasctime_s
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime_s, _localtime32_s, _localtime64_s
_mkgmtime, _mkgmtime32, _mkgmtime64
mktime, _mktime32, _mktime64
time, _time32, _time64

_heapchk

11/8/2018 • 2 minutes to read • Edit Online

Runs consistency checks on the heap.

Syntax

```
int _heapchk( void );
```

Return Value

_heapchk returns one of the following integer manifest constants defined in Malloc.h.

RETURN VALUE	CONDITION
_HEAPBADBEGIN	Initial header information is bad or cannot be found.
_HEAPBADNODE	Bad node has been found or heap is damaged.
_HEAPBADPTR	Pointer into heap is not valid.
_HEAPEMPTY	Heap has not been initialized.
_HEAPOK	Heap appears to be consistent.

In addition, if an error occurs, **_heapchk** sets **errno** to **ENOSYS**.

Remarks

The **_heapchk** function helps debug heap-related problems by checking for minimal consistency of the heap. If the operating system does not support **_heapchk**(for example, Windows 98), the function returns **_HEAPOK** and sets **errno** to **ENOSYS**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_heapchk	<malloc.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_heapchk.c
// This program checks the heap for
// consistency and prints an appropriate message.

#include <malloc.h>
#include <stdio.h>

int main( void )
{
   int  heapstatus;
   char *buffer;

   // Allocate and deallocate some memory
   if( (buffer = (char *)malloc( 100 )) != NULL )
      free( buffer );

   // Check heap status
   heapstatus = _heapchk();
   switch( heapstatus )
   {
   case _HEAPOK:
      printf(" OK - heap is fine\n" );
      break;
   case _HEAPEMPTY:
      printf(" OK - heap is empty\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
   }
}
```

```
OK - heap is fine
```

See also

Memory Allocation
_heapadd
_heapmin
_heapset
_heapwalk

_heapmin

10/31/2018 • 2 minutes to read • Edit Online

Releases unused heap memory to the operating system.

Syntax

```
int _heapmin( void );
```

Return Value

If successful, **_heapmin** returns 0; otherwise, the function returns -1 and sets **errno** to **ENOSYS**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_heapmin** function minimizes the heap by releasing unused heap memory to the operating system. If the operating system does not support **_heapmin**(for example, Windows 98), the function returns -1 and sets **errno** to **ENOSYS**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_heapmin	<malloc.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

Memory Allocation
free
_heapadd
_heapchk
_heapset
_heapwalk
malloc

_heapwalk

10/31/2018 • 3 minutes to read • Edit Online

Traverses the heap and returns information about the next entry.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime except in Debug builds. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _heapwalk( _HEAPINFO *entryinfo );
```

Parameters

entryinfo
Buffer to contain heap information.

Return Value

_heapwalk returns one of the following integer manifest constants defined in Malloc.h.

RETURN VALUE	MEANING
_HEAPBADBEGIN	Initial header information invalid or not found.
_HEAPBADNODE	Heap damaged or bad node found.
_HEAPBADPTR	The **_pentry** field of the **_HEAPINFO** structure does not contain a valid pointer into the heap or *entryinfo* is a null pointer.
_HEAPEND	End of the heap reached successfully.
_HEAPEMPTY	Heap not initialized.
_HEAPOK	No errors so far; *entryinfo* is updated with information about the next heap entry.

In addition, if an error occurs, **_heapwalk** sets **errno** to **ENOSYS**.

Remarks

The **_heapwalk** function helps debug heap-related problems in programs. The function walks through the heap, traversing one entry per call, and returns a pointer to a structure of type **_HEAPINFO** that contains information about the next heap entry. The **_HEAPINFO** type, defined in Malloc.h, contains the following elements.

FIELD	MEANING
`int *_pentry`	Heap entry pointer.
`size_t _size`	Size of the heap entry.
`int _useflag`	Flag that indicates whether the heap entry is in use.

A call to **_heapwalk** that returns **_HEAPOK** stores the size of the entry in the **_size** field and sets the **_useflag** field to either **_FREEENTRY** or **_USEDENTRY** (both are constants defined in Malloc.h). To obtain this information about the first entry in the heap, pass **_heapwalk** a pointer to a **_HEAPINFO** structure whose **_pentry** member is **NULL**. If the operating system does not support **_heapwalk**(for example, Windows 98), the function returns **_HEAPEND** and sets **errno** to **ENOSYS**.

This function validates its parameter. If *entryinfo* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_HEAPBADPTR**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_heapwalk	<malloc.h>	<errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_heapwalk.c

// This program "walks" the heap, starting
// at the beginning (_pentry = NULL). It prints out each
// heap entry's use, location, and size. It also prints
// out information about the overall state of the heap as
// soon as _heapwalk returns a value other than _HEAPOK
// or if the loop has iterated 100 times.

#include <stdio.h>
#include <malloc.h>

void heapdump(void);

int main(void)
{
    char *buffer;

    heapdump();
    if((buffer = (char *)malloc(59)) != NULL)
    {
        heapdump();
        free(buffer);
    }
    heapdump();
}

void heapdump(void)
{
    _HEAPINFO hinfo;
    int heapstatus;
    int numLoops;
    hinfo._pentry = NULL;
    numLoops = 0;
    while((heapstatus = _heapwalk(&hinfo)) == _HEAPOK &&
          numLoops < 100)
    {
        printf("%8s block at %Fp of size %4.4X\n",
               (hinfo._useflag == _USEDENTRY ? "USED" : "FREE"),
               hinfo._pentry, hinfo._size);
        numLoops++;
    }

    switch(heapstatus)
    {
    case _HEAPEMPTY:
        printf("OK - empty heap\n");
        break;
    case _HEAPEND:
        printf("OK - end of heap\n");
        break;
    case _HEAPBADPTR:
        printf("ERROR - bad pointer to heap\n");
        break;
    case _HEAPBADBEGIN:
        printf("ERROR - bad start of heap\n");
        break;
    case _HEAPBADNODE:
        printf("ERROR - bad node in heap\n");
        break;
    }
}
```

```
    USED block at 00310650 of size 0100
    USED block at 00310758 of size 0800
    USED block at 00310F60 of size 0080
    FREE block at 00310FF0 of size 0398
    USED block at 00311390 of size 000D
    USED block at 003113A8 of size 00B4
    USED block at 00311468 of size 0034
    USED block at 003114A8 of size 0039
...
    USED block at 00312228 of size 0010
    USED block at 00312240 of size 1000
    FREE block at 00313250 of size 1DB0
OK - end of heap
```

See also

Memory Allocation
_heapadd
_heapchk
_heapmin
_heapset

hypot, hypotf, hypotl, _hypot, _hypotf, _hypotl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the hypotenuse.

Syntax

```
double hypot(
   double x,
   double y
);
float hypotf(
   float x,
   float y
);
long double hypotl(
   long double x,
   long double y
);
double _hypot(
   double x,
   double y
);
float _hypotf(
   float x,
   float y
);
long double _hypotl(
   long double x,
   long double y
);
```

Parameters

x, *y*
Floating-point values.

Return Value

If successful, **hypot** returns the length of the hypotenuse; on overflow, **hypot** returns INF (infinity) and the **errno** variable is set to **ERANGE**. You can use **_matherr** to modify error handling.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **hypot** functions calculate the length of the hypotenuse of a right triangle, given the length of the two sides *x* and *y* (in other words, the square root of $x^2 + y^2$).

The versions of the functions that have leading underscores are provided for compatibility with earlier standards. Their behavior is identical to the versions that don't have leading underscores. We recommend using the versions without leading underscores for new code.

Requirements

ROUTINE	REQUIRED HEADER
hypot, **hypotf**, **hypotl**, **_hypot**, **_hypotf**, **_hypotl**	<math.h>

For more compatibility information, see Compatibility.

Example

```
// crt_hypot.c
// This program prints the hypotenuse of a right triangle.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 3.0, y = 4.0;

   printf( "If a right triangle has sides %2.1f and %2.1f, "
           "its hypotenuse is %2.1f\n", x, y, _hypot( x, y ) );
}
```

```
If a right triangle has sides 3.0 and 4.0, its hypotenuse is 5.0
```

See also

Floating-Point Support
_cabs
_matherr

ilogb, ilogbf, ilogbl

11/9/2018 • 2 minutes to read • Edit Online

Retrieves an integer that represents the unbiased base-2 exponent of the specified value.

Syntax

```
int ilogb(
   double x
);

int ilogb(
   float x
); //C++ only

int ilogb(
   long double x
); //C++ only

int ilogbf(
   float x
);

int ilogbl(
   long double x
);
```

Parameters

x
The specified value.

Return Value

If successful, return the base-2 exponent of *x* as a signed **int** value.

Otherwise, returns one of the following values, defined in <math.h>:

INPUT	RESULT
±0	FP_ILOGB0
±inf, ±nan, indefinite	FP_ILOGBNAN

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **ilogb** that take and return **float** and **long double** types. In a C program, **ilogb** always takes and returns a **double**.

Calling this function is similar to calling the equivalent **logb** function, then casting the return value to **int**.

Requirements

ROUTINE	C HEADER	C++ HEADER
ilogb, **ilogbf**, **ilogbl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
frexp
logb, logbf, logbl, _logb, _logbf

imaxabs

10/31/2018 • 2 minutes to read • Edit Online

Calculates the absolute value of an integer of any size.

Syntax

```
intmax_t imaxabs(
   intmax_t n
);
```

Parameters

n
Integer value.

Return Value

The **imaxabs** function returns the absolute value of the argument. There is no error return.

> **NOTE**
> Because the range of negative integers that can be represented by using **intmax_t** is larger than the range of positive integers that can be represented, it's possible to supply an argument to **imaxabs** that can't be converted. If the absolute value of the argument cannot be represented by the return type, the behavior of **imaxabs** is undefined.

Requirements

ROUTINE	REQUIRED HEADER
imaxabs	<inttypes.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_imaxabs.c
// Build using: cl /W3 /Tc crt_imaxabs.c
// This example calls imaxabs to compute an
// absolute value, then displays the results.

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int main(int argc, char *argv[])
{
   intmax_t x = LLONG_MIN + 2;

   printf("The absolute value of %lld is %lld\n", x, imaxabs(x));
}
```

```
The absolute value of -9223372036854775806 is 9223372036854775806
```

See also

Data Conversion
Floating-Point Support
abs, labs, llabs, _abs64
_cabs
fabs, fabsf, fabsl

imaxdiv

10/31/2018 • 2 minutes to read • Edit Online

Computes the quotient and the remainder of two integer values of any size as a single operation.

Syntax

```
imaxdiv_t imaxdiv(
   intmax_t numer,
   intmax_t denom
);
```

Parameters

numer
The numerator.

denom
The denominator.

Return Value

imaxdiv called with arguments of type intmax_t returns a structure of type imaxdiv_t that comprises the quotient and the remainder.

Remarks

The **imaxdiv** function divides *numer* by *denom* and thereby computes the quotient and the remainder. The **imaxdiv_t** structure contains the quotient, **intmax_t quot**, and the remainder, **intmax_t rem**. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program terminates with an error message.

Requirements

ROUTINE	REQUIRED HEADER
imaxdiv	<inttypes.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_imaxdiv.c
// Build using: cl /W3 /Tc crt_imaxdiv.c
// This example takes two integers as command-line
// arguments and calls imaxdiv to divide the first
// argument by the second, then displays the results.

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int main(int argc, char *argv[])
{
   intmax_t x,y;
   imaxdiv_t div_result;

   x = atoll(argv[1]);
   y = atoll(argv[2]);

   printf("The call to imaxdiv(%lld, %lld)\n", x, y);
   div_result = imaxdiv(x, y);
   printf("results in a quotient of %lld, and a remainder of %lld\n\n",
          div_result.quot, div_result.rem);
}
```

When built and then called with command line parameters of `9460730470000000 8766`, the code generates this output:

```
The call to imaxdiv(9460730470000000, 8766)
results in a quotient of 1079252848505, and a remainder of 5170
```

See also

Floating-Point Support
div
ldiv, lldiv

_initterm, _initterm_e

10/31/2018 • 2 minutes to read • Edit Online

Internal methods that walk a table of function pointers and initialize them.

The first pointer is the starting location in the table and the second pointer is the ending location.

Syntax

```
void __cdecl _initterm(
   PVFV *,
   PVFV *
);

int __cdecl _initterm_e(
   PVFV *,
   PVFV *
);
```

Return Value

A non-zero error code if an initialization fails and throws an error; 0 if no error occurs.

Remarks

These methods are only called internally during the initialization of a C++ program. Do not call these methods in a program.

When these methods walk a table of function entries, they skip **NULL** entries and continue.

See also

Alphabetical Function Reference

_invalid_parameter, _invalid_parameter_noinfo, _invalid_parameter_noinfo_noreturn, _invoke_watson

10/31/2018 • 2 minutes to read • Edit Online

These functions are used by the C Runtime Library to handle non-valid parameters passed to CRT Library functions. Your code may also use these functions to support default or customizable handling of non-valid parameters.

Syntax

```
extern "C" void __cdecl
_invalid_parameter(
    wchar_t const* const expression,
    wchar_t const* const function_name,
    wchar_t const* const file_name,
    unsigned int   const line_number,
    uintptr_t      const reserved);

extern "C" void __cdecl
_invalid_parameter_noinfo(void);

extern "C" __declspec(noreturn) void __cdecl
_invalid_parameter_noinfo_noreturn(void);

extern "C" __declspec(noreturn) void __cdecl
_invoke_watson(
    wchar_t const* const expression,
    wchar_t const* const function_name,
    wchar_t const* const file_name,
    unsigned int   const line_number,
    uintptr_t      const reserved);
```

Parameters

expression
A string representing the source code parameter expression that is not valid.

function_name
The name of the function that called the handler.

file_name
The source code file where the handler was called.

line_number
The line number in the source code where the handler was called.

reserved
Unused.

Return Value

These functions do not return a value. The **_invalid_parameter_noinfo_noreturn** and **_invoke_watson** functions do not return to the caller, and in some cases, **_invalid_parameter** and **_invalid_parameter_noinfo**

may not return to the caller.

Remarks

When C runtime library functions are passed non-valid parameters, the library functions call an *invalid parameter handler*, a function that may be specified by the programmer to do any of several things. For example, it may report the issue to the user, write to a log, break in a debugger, terminate the program, or do nothing at all. If no function is specified by the programmer, a default handler, **_invoke_watson**, is called.

By default, when a non-valid parameter is identified in debug code, CRT library functions call the function **_invalid_parameter** using verbose parameters. In non-debug code, the **_invalid_parameter_noinfo** function is called, which calls the **_invalid_parameter** function using empty parameters. If the non-debug CRT library function requires program termination, the **_invalid_parameter_noinfo_noreturn** function is called, which calls the **_invalid_parameter** function using empty parameters, followed by a call to the **_invoke_watson** function to force program termination.

The **_invalid_parameter** function checks whether a user-defined invalid parameter handler was set, and if so, calls it. For example, if a user-defined thread-local handler was set by a call to set_thread_local_invalid_parameter_handler in the current thread, it is called, then the function returns. Otherwise, if a user-defined global invalid parameter handler was set by a call to set_invalid_parameter_handler, it is called, then the function returns. Otherwise, the default handler **_invoke_watson** is called. The default behavior of **_invoke_watson** is to terminate the program. User-defined handlers may terminate or return. We recommend that user-defined handlers terminate the program unless recovery is certain.

When the default handler **_invoke_watson** is called, if the processor supports a __fastfail operation, it is invoked using a parameter of **FAST_FAIL_INVALID_ARG** and the process terminates. Otherwise, a fast fail exception is raised, which can be caught by an attached debugger. If the process is allowed to continue, it is terminated by a call to the Windows **TerminateProcess** function using an exception code status of **STATUS_INVALID_CRUNTIME_PARAMETER**.

Requirements

FUNCTION	REQUIRED HEADER
_invalid_parameter, **_invalid_parameter_noinfo**, **_invalid_parameter_noinfo_noreturn**, **_invoke_watson**	<corecrt.h>

For additional compatibility information, see Compatibility.

See also

AIphabetical Function Reference
_get_invalid_parameter_handler, _get_thread_local_invalid_parameter_handler
_set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler
Parameter Validation

isalnum, iswalnum, _isalnum_l, _iswalnum_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents an alphanumeric character.

Syntax

```
int isalnum( int c );
int iswalnum( wint_t c );
int _isalnum_l( int c,  _locale_t locale );
int _iswalnum_l( wint_t c, _locale_t locale );
```

Parameters

c
Integer to test.

locale
The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an alphanumeric character. **isalnum** returns a nonzero value if either **isalpha** or **isdigit** is nonzero for *c*, that is, if *c* is within the ranges A - Z, a - z, or 0 - 9. **iswalnum** returns a nonzero value if either **iswalpha** or **iswdigit** is nonzero for *c*. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale parameter that's passed in instead of the current locale. For more information, see Locale.

The behavior of **isalnum** and **_isalnum_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istalnum	**isalnum**	_ismbcalnum	**iswalnum**
_istalnum_l	**_isalnum_l**	**_ismbcalnum_l**	**_iswalnum_l**

Requirements

ROUTINE	REQUIRED HEADER
isalnum	<ctype.h>
iswalnum	<ctype.h> or <wchar.h>
_isalnum_l	<ctype.h>

ROUTINE	REQUIRED HEADER
_iswalnum_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isalpha, iswalpha, _isalpha_l, _iswalpha_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether an integer represents an alphabetic character.

Syntax

```
int isalpha(
   int c
);
int iswalpha(
   wint_t c
);
int _isalpha_l(
   int c,
   _locale_t locale
);
int _iswalpha_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
The locale to use instead of the current locale.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an alphabetic character. **isalpha** returns a nonzero value if *c* is within the ranges A - Z or a - z. **iswalpha** returns a nonzero value only for wide characters for which iswupper or **iswlower** is nonzero; that is, for any wide character that is one of an implementation-defined set for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale parameter that's passed in instead of the current locale. For more information, see Locale.

The behavior of **isalpha** and **_isalpha_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istalpha	**isalpha**	**_ismbcalpha**	**iswalpha**
_istalpha_l	**_isalpha_l**	**_ismbcalpha_l**	**_iswalpha_l**

Requirements

ROUTINE	REQUIRED HEADER
isalpha	<ctype.h>
iswalpha	<ctype.h> or <wchar.h>
_isalpha_l	<ctype.h>
_iswalpha_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isascii, __isascii, iswascii

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a particular character is an ASCII character.

Syntax

```
int __isascii(
   int c
);
int iswascii(
   wint_t c
);

#define isascii __isascii
```

Parameters

c
Integer to test.

Return Value

Each of these routines returns nonzero if **c** is a particular representation of an ASCII character. **__isascii** returns a nonzero value if **c** is an ASCII character (in the range 0x00 - 0x7F). **iswascii** returns a nonzero value if **c** is a wide-character representation of an ASCII character. Each of these routines returns 0 if **c** does not satisfy the test condition.

Remarks

Both **__isascii** and **iswascii** are implemented as macros unless the preprocessor macro _CTYPE_DISABLE_MACROS is defined.

For backward compatibility, **isascii** is implemented as a macro only if __STDC__ is not defined or is defined as 0; otherwise it is undefined.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istascii	**__isascii**	**__isascii**	**iswascii**

Requirements

ROUTINE	REQUIRED HEADER
isascii, **__isascii**	C: <ctype.h> C++: <cctype> or <ctype.h>

ROUTINE	REQUIRED HEADER
iswascii	C: <wctype.h>, <ctype.h>, or <wchar.h> C++: <cwctype>, <cctype>, <wctype.h>, <ctype.h>, or <wchar.h>

The **isascii**, **__isascii** and **iswascii** functions are Microsoft specific. For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isatty

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _isatty instead.

_isatty

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a file descriptor is associated with a character device.

Syntax

```
int _isatty( int fd );
```

Parameters

fd
File descriptor that refers to the device to be tested.

Return Value

_isatty returns a nonzero value if the descriptor is associated with a character device. Otherwise, **_isatty** returns 0.

Remarks

The **_isatty** function determines whether *fd* is associated with a character device (a terminal, console, printer, or serial port).

This function validates the *fd* parameter. If *fd* is a bad file pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EBADF**.

Requirements

ROUTINE	REQUIRED HEADER
_isatty	<io.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_isatty.c
/* This program checks to see whether
* stdout has been redirected to a file.
*/

#include <stdio.h>
#include <io.h>

int main( void )
{
   if( _isatty( _fileno( stdout ) ) )
      printf( "stdout has not been redirected to a file\n" );
   else
      printf( "stdout has been redirected to a file\n");
}
```

Sample Output

```
stdout has not been redirected to a file
```

See also

File Handling

isblank, iswblank, _isblank_l, _iswblank_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether an integer represents a blank character.

Syntax

```
int isblank(
   int c
);
int iswblank(
   wint_t c
);
int _isblank_l(
   int c,
   _locale_t locale
);
int _iswblank_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a space or horizontal tab character, or is one of a locale-specific set of characters that are used to separate words within a line of text. **isblank** returns a nonzero value if *c* is a space character (0x20) or horizontal tab character (0x09). The result of the test condition for the **isblank** functions depends on the **LC_CTYPE** category setting of the locale; for more information, see setlocale, _wsetlocale. The versions of these functions that do not have the **_l** suffix use the current locale for any locale-dependent behavior; the versions that do have the **_l** suffix are identical except that they use the locale that's passed in instead. For more information, see Locale.

iswblank returns a nonzero value if *c* is a wide character that corresponds to a standard space or horizontal tab character.

The behavior of **isblank** and **_isblank_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istblank	**isblank**	_ismbcblank	**iswblank**
_istblank_l	**_isblank_l**	_ismbcblank_l	**_iswblank_l**

Requirements

ROUTINE	REQUIRED HEADER
isblank	<ctype.h>
iswblank	<ctype.h> or <wchar.h>
_isblank_l	<ctype.h>
_iswblank_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

iscntrl, iswcntrl, _iscntrl_l, _iswcntrl_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether an integer represents a control character.

Syntax

```
int iscntrl(
   int c
);
int iswcntrl(
   wint_t c
);
int _iscntrl_l(
   int c,
   _locale_t locale
);
int _iswcntrl_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test

locale
The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a control character. **iscntrl** returns a nonzero value if *c* is a control character (0x00 - 0x1F or 0x7F). **iswcntrl** returns a nonzero value if *c* is a control wide character. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale parameter that's passed in instead of the current locale. For more information, see Locale.

The behavior of **iscntrl** and **_iscntrl_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istcntrl	**iscntrl**	**iscntrl**	**iswcntrl**
_istcntrl_l	**_iscntrl_l**	**_iscntrl_l**	**_iswcntrl_l**

Requirements

ROUTINE	REQUIRED HEADER
iscntrl	<ctype.h>
iswcntrl	<ctype.h> or <wchar.h>
_iscntrl_l	<ctype.h>
_iswcntrl_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

_isctype, iswctype, _isctype_l, _iswctype_l

10/31/2018 • 2 minutes to read • Edit Online

Tests *c* for the ctype property specified by the *desc* argument. For each valid value of *desc*, there is an equivalent wide-character classification routine.

Syntax

```
int _isctype(
   int c,
   _ctype_t desc
);
int _isctype_l(
   int c,
   _ctype_t desc,
   _locale_t locale
);
int iswctype(
   wint_t c,
   wctype_t desc
);
int _iswctype_l(
   wint_t c,
   wctype_t desc,
   _locale_t locale
);
```

Parameters

c
Integer to test.

desc
Property to test for. This is normally retrieved using ctype or wctype.

locale
The locale to use for any locale-dependent tests.

Return Value

_isctype and **iswctype** return a nonzero value if *c* has the property specified by *desc* in the current locale or 0 if it does not. The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **_isctype** and **_isctype_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
n/a	**_isctype**	n/a	**_iswctype**
n/a	**_isctype_l**	n/a	**_iswctype_l**

Requirements

ROUTINE	REQUIRED HEADER
_isctype	<ctype.h>
iswctype	<ctype.h> or <wchar.h>
_isctype_l	<ctype.h>
_iswctype_l	<ctype.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Character Classification
Locale
is, isw Routines

iscsym, iscsymf, __iscsym, __iswcsym, __iscsymf, __iswcsymf, _iscsym_l, _iswcsym_l, _iscsymf_l, _iswcsymf_l

10/31/2018 • 2 minutes to read • Edit Online

Determine if an integer represents a character that may be used in an identifier.

Syntax

```
int __iscsym(
   int c
);
int __iswcsym(
   wint_t c
);
int __iscsymf(
   int c
);
int __iswcsymf(
   wint_t c
);
int _iscsym_l(
   int c,
   _locale_t locale
);
int _iswcsym_l(
   wint_t c,
   _locale_t locale
);
int _iscsymf_l(
   int c,
   _locale_t locale
);
int _iswcsymf_l(
   wint_t c,
   _locale_t locale
);
#define iscsym __iscsym
#define iscsymf __iscsymf
```

Parameters

c
Integer to test. *c* should be in the range of 0-255 for the narrow character version of the function.

locale
The locale to use.

Return Value

Both **__iscsym** and **__iswcsym** return a nonzero value if *c* is a letter, underscore, or digit. Both **__iscsymf** and **__iswcsymf** return a nonzero value if *c* is a letter or an underscore. Each of these routines returns 0 if *c* does not satisfy the test condition. The versions of these functions with the **_l** suffix are identical except that they use the *locale* passed in instead of the current locale for their locale-dependent behavior. For more information, see

Locale.

Remarks

These routines are defined as macros unless the preprocessor macro _CTYPE_DISABLE_MACROS is defined. When you use the macro versions of these routines, the arguments can be evaluated more than once. Be careful when you use expressions that have side effects within the argument list.

For backward compatibility, **iscsym** and **iscsymf** are defined as macros only when __STDC__ is not defined or is defined as 0; otherwise they are undefined.

Requirements

ROUTINE	REQUIRED HEADER
iscsym, **iscsymf**, **__iscsym**, **__iswcsym**, **__iscsymf**, **__iswcsymf**, **_iscsym_l**, **_iswcsym_l**, **_iscsymf_l**, **_iswcsymf_l**	C: <ctype.h> C++: <cctype> or <ctype.h>

The **iscsym**, **iscsymf**, **__iscsym**, **__iswcsym**, **__iscsymf**, **__iswcsymf**, **_iscsym_l**, **_iswcsym_l**, **_iscsymf_l**, and **_iswcsymf_l** routines are Microsoft specific. For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isdigit, iswdigit, _isdigit_l, _iswdigit_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents a decimal-digit character.

Syntax

```
int isdigit(
   int c
);
int iswdigit(
   wint_t c
);
int _isdigit_l(
   int c,
   _locale_t locale
);
int _iswdigit_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a decimal-digit character. **isdigit** returns a nonzero value if *c* is a decimal digit (0 - 9). **iswdigit** returns a nonzero value if *c* is a wide character that corresponds to a decimal-digit character. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale that's passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **isdigit** and **_isdigit_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istdigit	**isdigit**	_ismbcdigit	**iswdigit**
_istdigit_l	**_isdigit_l**	_ismbcdigit_l	**_iswdigit_l**

Requirements

ROUTINE	REQUIRED HEADER
isdigit	<ctype.h>
iswdigit	<ctype.h> or <wchar.h>
_isdigit_l	<ctype.h>
_iswdigit_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isfinite, _finite, _finitef

3/1/2019 • 2 minutes to read • Edit Online

Determines whether a floating-point value is finite.

Syntax

```
int isfinite(
   /* floating-point */ x
); /* C-only macro */

template <class FloatingType>
inline bool isfinite(
   FloatingType x
) throw(); /* C++-only template function */

int _finite(
   double x
);

int _finitef(
   float x
); /* x64 and ARM/ARM64 only */
```

Parameters

x
The floating-point value to test.

Return value

The `isfinite` macro and the `_finite` and `_finitef` functions return a non-zero value if *x* is either a normal or subnormal finite value. They return 0 if the argument is infinite or a NaN. The C++ inline template function `isfinite` behaves the same way, but returns **true** or **false**.

Remarks

`isfinite` is a macro when compiled as C, and an inline template function when compiled as C++. The `_finite` and `_finitef` functions are Microsoft-specific. The `_finitef` function is only available when compiled for x86, ARM, or ARM64 platforms.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
`_finite`	<float.h> or <math.h>	<float.h>, <math.h>, <cfloat>, or <cmath>
`isfinite`, `_finitef`	<math.h>	<math.h> or <cmath>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
fpclassify
_fpclass, _fpclassf
isinf
isnan, _isnan, _isnanf
isnormal

isgraph, iswgraph, _isgraph_l, _iswgraph_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether an integer represents a graphical character.

Syntax

```
int isgraph(
   int c
);
int iswgraph(
   wint_t c
);
int _isgraph_l(
   int c,
   _locale_t locale
);
int _iswgraph_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a printable character other than a space. **isgraph** returns a nonzero value if *c* is a printable character other than a space. **iswgraph** returns a nonzero value if *c* is a printable wide character other than a wide character space. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale that's passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **isgraph** and **_isgraph_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istgraph	**isgraph**	_ismbcgraph	**iswgraph**
_istgraph_l	**_isgraph_l**	_ismbcgraph_l	**_iswgraph_l**

Requirements

ROUTINE	REQUIRED HEADER
isgraph	<ctype.h>
iswgraph	<ctype.h> or <wchar.h>
_isgraph_l	<ctype.h>
_iswgraph_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered

2/4/2019 • 2 minutes to read • Edit Online

Determines the ordering relationship between two floating-point values.

Syntax

```
int isgreater(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */

int isgreaterequal(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */

int isless(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */

int islessequal(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */

int islessgreater(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */

int isunordered(
   /* floating-point */ x,
   /* floating-point */ y
); /* C-only macro */
```

```
template <class FloatingType1, class FloatingType2>
inline bool isgreater(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */

template <class FloatingType1, class FloatingType2>
inline bool isgreaterequal(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */

template <class FloatingType1, class FloatingType2>
inline bool isless(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */

template <class FloatingType1, class FloatingType2>
inline bool islessequal(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */

template <class FloatingType1, class FloatingType2>
inline bool islessgreater(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */

template <class FloatingType1, class FloatingType2>
inline bool isunordered(
   FloatingType1 x,
   FloatingType2 y
) throw(); /* C++-only template function */
```

Parameters

x, *y*
The floating-point values to compare.

Return Value

In all comparisons, infinities of the same sign compare as equal. Negative infinity is less than any finite value or positive infinity. Positive infinity is greater than any finite value or negative infinity. Zeroes are equal regardless of sign. NaNs are not less than, equal to, or greater than any value, including another NaN.

When neither argument is a NaN, the ordering macros **isgreater**, **isgreaterequal**, **isless**, and **islessequal** return a non-zero value if the specified ordering relation between *x* and *y* holds true. These macros return 0 if either or both arguments are NaNs or if the ordering relationship is false. The function forms behave the same way, but return **true** or **false**.

The **islessgreater** macro returns a non-zero value if both *x* and *y* are not NaNs, and *x* is either less than or greater than *y*. It returns 0 if either or both arguments are NaNs, or if the values are equal. The function form behaves the same way, but returns **true** or **false**.

The **isunordered** macro returns a non-zero value if either *x*, *y*, or both are NaNs. Otherwise, it returns 0. The function form behaves the same way, but returns **true** or **false**.

Remarks

These comparison operations are implemented as macros when compiled as C, and as inline template functions

when compiled as C++.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
isgreater, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isunordered**	<math.h>	<math.h> or <cmath>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
isfinite, _finite, _finitef
isinf
isnan, _isnan, _isnanf
_fpclass, _fpclassf

isinf

2/4/2019 • 2 minutes to read • Edit Online

Determines whether a floating-point value is an infinity.

Syntax

```
int isinf(
   /* floating-point */ x
); /* C-only macro */

template <class FloatingType>
inline bool isinf(
   FloatingType x
) throw(); /* C++-only template function */
```

Parameters

x
The floating-point value to test.

Return value

isinf returns a nonzero value (**true** in C++ code) if the argument *x* is a positive or negative infinity. **isinf** returns 0 (**false** in C++ code) if the argument is finite or a NAN. Both normal and subnormal floating-point values are considered finite.

Remarks

isinf is a macro when compiled as C, and an inline template function when compiled as C++.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
isinf	<math.h>	<math.h> or <cmath>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
fpclassify
_fpclass, _fpclassf
isfinite, _finite, _finitef
isnan, _isnan, _isnanf
isnormal

isleadbyte, _isleadbyte_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a character is the lead byte of a multibyte character.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int isleadbyte( int c );
int _isleadbyte_l( int c );
```

Parameters

c
Integer to test.

Return Value

isleadbyte returns a nonzero value if the argument satisfies the test condition or 0 if it does not. In the "C" locale and in single-byte character set (SBCS) locales, **isleadbyte** always returns 0.

Remarks

The **isleadbyte** macro returns a nonzero value if its argument is the first byte of a multibyte character. **isleadbyte** produces a meaningful result for any integer argument from -1 (**EOF**) to **UCHAR_MAX** (0xFF), inclusive.

The expected argument type of **isleadbyte** is **int**; if a signed character is passed, the compiler may convert it to an integer by sign extension, yielding unpredictable results.

The version of this function with the **_l** suffix is identical except that it uses the locale passed in instead of the current locale for its locale-dependent behavior.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istleadbyte	Always returns false	**_isleadbyte**	Always returns false

Requirements

ROUTINE	REQUIRED HEADER
isleadbyte	<ctype.h>
_isleadbyte_l	<ctype.h>

For additional compatibility information, see Compatibility.

See also

Byte Classification
Locale
_ismbb Routines

islower, iswlower, _islower_l, _iswlower_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents a lowercase character.

Syntax

```
int islower(
   int c
);
int iswlower(
   wint_t c
);
int islower_l(
   int c,
   _locale_t locale
);
int _iswlower_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a lowercase character. **islower** returns a nonzero value if *c* is a lowercase character (a - z). **iswlower** returns a nonzero value if *c* is a wide character that corresponds to a lowercase letter, or if *c* is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale that's passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **islower** and **_islower_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istlower	**islower**	_ismbclower	**iswlower**
_istlower_l	`_islower _l`	_ismbclower_l	**_liswlower_l**

Requirements

ROUTINE	REQUIRED HEADER
islower	<ctype.h>
iswlower	<ctype.h> or <wchar.h>
_islower_l	<ctype.h>
_swlower_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

_ismbbalnum, _ismbbalnum_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a specified multibyte character is alpha or numeric.

Syntax

```
int _ismbbalnum(
   unsigned int c
);
int _ismbbalnum_l(
   unsigned int c
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbalnum returns a nonzero value if the expression:

`isalnum(c) || _ismbbkalnum(c)`

is nonzero for *c*, or 0 if it is not.

The version of this function with the **_l** suffix is identical except that it uses the locale passed in instead of the current locale for its locale-dependent behavior.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbalnum	<mbctype.h>
_ismbbalnum_l	<mbctype.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Byte Classification
_ismbb Routines

_ismbbalpha, _ismbbalpha_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a specified multibyte character is alpha.

Syntax

```
int _ismbbalpha(
   unsigned int c
);
int _ismbbalpha_l(
   unsigned int c
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbalpha returns a nonzero value if the expression:

```
isalpha(c) || _ismbbkalnum(c)
```

is nonzero for *c*, or 0 if it is not. **_ismbbalpha** uses the current locale for any locale-dependent character settings. **_ismbbalpha_l** is identical except that it uses the locale passed in.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbalpha	<mbctype.h>
_ismbbalpha_l	<mbctype.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Byte Classification
_ismbb Routines

_ismbbblank, _ismbbblank_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a specified multibyte character is a blank character.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbbblank(
   unsigned int c
);
int _ismbbblank_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbblank returns a nonzero value if *c* represents a space (0x20) character, a horizontal tab (0x09) character, or a locale-specific character that's used to separate words within a line of text for which **isspace** is true; otherwise, returns 0. **_ismbbblank** uses the current locale for any locale-dependent behavior. **_ismbbblank_l** is identical except that it instead uses the locale that's passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbblank	<mbctype.h>
_ismbbblank_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbgraph, _ismbbgraph_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a particular multibyte character is a graphical character.

Syntax

```
int _ismbbgraph (
   unsigned int c
);
int _ismbbgraph_l (
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

Returns a nonzero value if the expression:

```
isctype(c, ( _PUNCT | _UPPER | _LOWER | _DIGIT )) || _ismbbkprint(c)
```

is nonzero for *c*, or 0 if it is not. **_ismbbgraph** uses the current locale for any locale-dependent behavior. **_ismbbgraph_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbgraph	<mbctype.h>
_ismbbgraph_l	<mbctype.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Byte Classification
_ismbb Routines

_ismbbkalnum, _ismbbkalnum_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a particular multibyte character is a non-ASCII text symbol.

Syntax

```
int _ismbbkalnum(
   unsigned int c
);
int _ismbbkalnum_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbkalnum returns a nonzero value if the integer *c* is a non-ASCII text symbol other than punctuation, or 0 if it is not. **_ismbbkalnum** uses the current locale for locale-dependent character information. **_ismbbkalnum_l** is identical to **_ismbbkalnum** except that it takes the locale as a parameter. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbkalnum	<mbctype.h>
_ismbbkalnum_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbkana, _ismbbkana_l

10/31/2018 • 2 minutes to read • Edit Online

Tests for a katakana symbol and is specific to code page 932.

Syntax

```
int _ismbbkana(
   unsigned int c
);
int _ismbbkana_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbkana returns a nonzero value if the integer *c* is a katakana symbol or 0 if it is not. **_ismbbkana** uses the current locale for locale-dependent character information. **_ismbbkana_l** is identical except that it uses the locale object passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbkana	<mbctype.h>
_ismbbkana_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbkprint, _ismbbkprint_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a particular multibyte character is a punctuation symbol.

Syntax

```
int _ismbbkprint(
   unsigned int c
);
int _ismbbkprint_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbkprint returns a nonzero value if the integer *c* is a non-ASCII text or non-ASCII punctuation symbol or 0 if it is not. For example, in code page 932 only, **_ismbbkprint** tests for katakana alphanumeric or katakana punctuation (range: 0xA1 - 0xDF). **_ismbbkprint** uses the current locale for locale-dependent character settings. **_ismbbkprint_l** is identical except that it uses the locale passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbkprint	<mbctype.h>
_ismbbkprint_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbkpunct, _ismbbkpunct_l

10/31/2018 • 2 minutes to read • Edit Online

Checks whether a multibyte character is a punctuation character.

Syntax

```
int _ismbbkpunct(
   unsigned int c
);
int _ismbbkpunct_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbkpunct returns a nonzero value if the integer *c* is a non-ASCII punctuation symbol, or 0 if it is not. For example, in code page 932 only, **_ismbbkpunct** tests for katakana punctuation. **_ismbbkpunct** uses the current locale for any locale-dependent character settings. **_ismbbkpunct_l** is identical except that it uses the locale that's passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbkpunct	<mbctype.h>
_ismbbkpunct_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbblead, _ismbblead_l

10/31/2018 • 2 minutes to read • Edit Online

Tests a character to determine whether it is a lead byte of a multibyte character.

Syntax

```
int _ismbblead(
   unsigned int c
);
int _ismbblead_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

Returns a nonzero value if the integer *c* is the first byte of a multibyte character.

Remarks

Multibyte characters consist of a lead byte followed by a trailing byte. Lead bytes are distinguished by being in a particular range for a given character set. For example, in code page 932 only, lead bytes range from 0x81 - 0x9F and 0xE0 - 0xFC.

_ismbblead uses the current locale for locale-dependent behavior. **_ismbblead_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istlead	Always returns false	**_ismbblead**	Always returns false

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_ismbblead	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>
_ismbblead_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbprint, _ismbbprint_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a specified multibyte character is a print character.

Syntax

```
int _ismbbprint(
   unsigned int c
);
int _ismbbprint_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbprint returns a nonzero value if the expression:

```
isprint(c) || _ismbbkprint(c)
```

is nonzero for *c*, or 0 if it is not. **_ismbbprint** uses the current locale for any locale-dependent behavior. **_ismbbprint_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbprint	<mbctype.h>
_ismbbprint_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbpunct, _ismbbpunct_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a particular character is a punctuation character.

Syntax

```
int _ismbbpunct(
   unsigned int c
);
int _ismbbpunct_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Integer to be tested.

locale
Locale to use.

Return Value

_ismbbpunct returns a nonzero value if the integer *c* is a non-ASCII punctuation symbol. **_ismbbpunct** uses the current locale for any locale-dependent character settings. **_ismbbpunct_l** is identical except that it uses the locale that's passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_ismbbpunct	<mbctype.h>
_ismbbpunct_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbbtrail, _ismbbtrail_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether a byte is a trailing byte of a multibyte character.

Syntax

```
int _ismbbtrail(
   unsigned int c
);
int _ismbbtrail_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
The integer to be tested.

locale
The locale to use.

Return Value

_ismbbtrail returns a nonzero value if the integer *c* is the second byte of a multibyte character. For example, in code page 932 only, valid ranges are 0x40 to 0x7E and 0x80 to 0xFC.

Remarks

_ismbbtrail uses the current locale for locale-dependent behavior. **_ismbbtrail_l** is identical except that it uses the locale that's passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_ismbbtrail	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>
_ismbbtrail_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility.

See also

Byte Classification
_ismbb Routines

_ismbcalnum, _ismbcalnum_l, _ismbcalpha, _ismbcalpha_l, _ismbcdigit, _ismbcdigit_l

10/31/2018 • 2 minutes to read • Edit Online

Checks whether a multibyte character is an alphanumeric, alpha, or digit character.

> **IMPORTANT**
>
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbcalnum
(
   unsigned int c
);
int _ismbcalnum_l
(
   unsigned int c,
   _locale_t locale
);
int _ismbcalpha
(
   unsigned int c
);
int _ismbcalpha_l
(
   unsigned int c,
   _locale_t locale
);
int _ismbcdigit
(
   unsigned int c
);
int _ismbcdigit_l
(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be tested.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these routines tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbcalnum, **_ismbcalnum_l**	Alphanumeric	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: See examples for **_ismbcdigit** and **_ismbcalpha**.
_ismbcalpha, **_ismbcalpha_l**	Alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII English letter: 0x41<=*c*<=0x5A or 0x61<=*c*<=0x7A; or a katakana letter: 0xA6<=*c*<=0xDF.
_ismbcdigit, **_ismbcdigit**	Digit	Returns nonzero if and only if *c* is a single-byte representation of an ASCII digit: 0x30<=*c*<=0x39.

Requirements

ROUTINE	REQUIRED HEADER
_ismbcalnum, **_ismbcalnum_l**	<mbstring.h>
_ismbcalpha, **_ismbcalpha_l**	<mbstring.h>
_ismbcdigit, **_ismbcdigit_l**	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbcgraph, _ismbcgraph_l, _ismbcprint, _ismbcprint_l, _ismbcpunct, _ismbcpunct_l, _ismbcblank, _ismbcblank_l, _ismbcspace, _ismbcspace_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether character is a graphical character, a display character, a punctuation character, or a space character.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbcgraph(
   unsigned int c
);
int _ismbcgraph_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcprint(
   unsigned int c
);
int _ismbcprint_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcpunct(
   unsigned int c
);
int _ismbcpunct_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcblank(
   unsigned int c
);
int _ismbcblank_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcspace(
   unsigned int c
);
int _ismbcspace_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be determined.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition, or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

The versions of these functions are identical, except that the ones that have the **_l** suffix use the locale that's passed in for their locale-dependent behavior, instead of the current locale. For more information, see Locale.

Remarks

Each of these functions tests a given multibyte character for a given condition.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbcgraph	Graphic	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character except a white space ( ).
_ismbcprint	Printable	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana printable character including a white space ( ).
_ismbcpunct	Punctuation	Returns nonzero if and only if *c* is a single-byte representation of any ASCII or katakana punctuation character.
_ismbcblank	Space or horizontal tab	Returns nonzero if and only if *c* is a space or horizontal tab character: *c*=0x20 or *c*=0x09.
_ismbcspace	White space	Returns nonzero if and only if *c* is a white-space character: *c*=0x20 or 0x09<=*c*<=0x0D.

Requirements

ROUTINE	REQUIRED HEADER
_ismbcgraph	<mbstring.h>
_ismbcgraph_l	<mbstring.h>
_ismbcprint	<mbstring.h>

ROUTINE	REQUIRED HEADER
_ismbcprint_l	<mbstring.h>
_ismbcpunct	<mbstring.h>
_ismbcpunct_l	<mbstring.h>
_ismbcblank	<mbstring.h>
_ismbcblank_l	<mbstring.h>
_ismbcspace	<mbstring.h>
_ismbcspace_l	<mbstring.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Character Classification
Locale
Interpretation of Multibyte-Character Sequences
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbchira, _ismbchira_l, _ismbckata, _ismbckata_l

10/31/2018 • 2 minutes to read • Edit Online

Code Page 932 Specific functions

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbchira(
   unsigned int c
);
int _ismbchira_l(
   unsigned int c,
   _locale_t locale
);
int _ismbckata(
   unsigned int c
);
int _ismbckata_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be tested.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

ROUTINE	TEST CONDITION (CODE PAGE 932 ONLY)
_ismbchira	Double-byte Hiragana: 0x829F<=*c*<=0x82F1.

ROUTINE	TEST CONDITION (CODE PAGE 932 ONLY)
_ismbchira_l	Double-byte Hiragana: 0x829F<=*c*<=0x82F1.
_ismbckata	Double-byte katakana: 0x8340<=*c*<=0x8396.
_ismbckata_l	Double-byte katakana: 0x8340<=*c*<=0x8396.

End Code Page 932 Specific

Requirements

ROUTINE	REQUIRED HEADER
_ismbchira	<mbstring.h>
_ismbchira_l	<mbstring.h>
_ismbckata	<mbstring.h>
_ismbckata_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
is, isw Routines
Locale
Interpretation of Multibyte-Character Sequences

_ismbcl0, _ismbcl0_l, _ismbcl1, _ismbcl1_l, _ismbcl2, _ismbcl2_l

10/31/2018 • 2 minutes to read • Edit Online

Code Page 932 Specific functions, using the current locale or a specified LC_CTYPE conversion state category.

> **IMPORTANT**
>
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbcl0(
   unsigned int c
);
int _ismbcl0_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcl1(
   unsigned int c
);
int _ismbcl1_l(
   unsigned int c ,
   _locale_t locale
);
int _ismbcl2(
   unsigned int c
);
int _ismbcl2_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be tested.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

ROUTINE	TEST CONDITION (CODE PAGE 932 ONLY)
_ismbcl0	JIS non-Kanji: 0x8140<=*c*<=0x889E.
_ismbcl0_l	JIS non-Kanji: 0x8140<=*c*<=0x889E.
_ismbcl1	JIS level-1: 0x889F<=*c*<=0x9872.
_ismbcl1_l	JIS level-1: 0x889F<=*c*<=0x9872.
_ismbcl2	JIS level-2: 0x989F<=*c*<=0xEAA4.
_ismbcl2_l	JIS level-2: 0x989F<=*c*<=0xEAA4.

The functions check that the specified value *c* matches the test conditions described above, but do not check that *c* is a valid multibyte character. If the lower byte is in the ranges 0x00 - 0x3F, 0x7F, or 0xFD - 0xFF, these functions return a nonzero value, indicating that the character satisfies the test condition. Use _ismbbtrail to test whether the multibyte character is defined.

End Code Page 932 Specific

Requirements

ROUTINE	REQUIRED HEADER
_ismbcl0	<mbstring.h>
_ismbcl0_l	<mbstring.h>
_ismbcl1	<mbstring.h>
_ismbcl1_l	<mbstring.h>
_ismbcl2	<mbstring.h>
_ismbcl2_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
is, isw Routines

_ismbclegal, _ismbclegal_l, _ismbcsymbol, _ismbcsymbol_l

10/31/2018 • 2 minutes to read • Edit Online

Checks whether a multibyte character is a legal or symbol character.

> **IMPORTANT**
>
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbclegal(
   unsigned int c
);
int _ismbclegal_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcsymbol(
   unsigned int c
);
int _ismbcsymbol_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be tested.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbclegal	Valid multibyte	Returns nonzero if and only if the first byte of *c* is within ranges 0x81 - 0x9F or 0xE0 - 0xFC, while the second byte is within ranges 0x40 - 0x7E or 0x80 - FC.
_ismbcsymbol	Multibyte symbol	Returns nonzero if and only if 0x8141<=*c*<=0x81AC.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istlegal	Always returns false	**_ismbclegal**	Always returns false.
_istlegal_l	Always returns false	**_ismbclegal_l**	Always returns false.

Requirements

ROUTINE	REQUIRED HEADER
_ismbclegal, **_ismbclegal_l**	<mbstring.h>
_ismbcsymbol, **_ismbcsymbol_l**	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l

10/31/2018 • 2 minutes to read • Edit Online

Checks whether a multibyte character is lowercase or uppercase.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbclower(
   unsigned int c
);
int _ismbclower_l(
   unsigned int c,
   _locale_t locale
);
int _ismbcupper(
   unsigned int c
);
int _ismbcupper_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to be tested.

locale
Locale to use.

Return Value

Each of these routines returns a nonzero value if the character satisfies the test condition or 0 if it does not. If *c* <= 255 and there is a corresponding **_ismbb** routine (for example, **_ismbcalnum** corresponds to **_ismbbalnum**), the result is the return value of the corresponding **_ismbb** routine.

Remarks

Each of these functions tests a given multibyte character for a given condition.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

ROUTINE	TEST CONDITION	CODE PAGE 932 EXAMPLE
_ismbclower	Lowercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII lowercase English letter: 0x61<=*c*<=0x7A.
_ismbclower_l	Lowercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII lowercase English letter: 0x61<=*c*<=0x7A.
_ismbcupper	Uppercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII uppercase English letter: 0x41<=*c*<=0x5A.
_ismbcupper_l	Uppercase alphabetic	Returns nonzero if and only if *c* is a single-byte representation of an ASCII uppercase English letter: 0x41<=*c*<=0x5A.

Requirements

ROUTINE	REQUIRED HEADER
_ismbclower	<mbstring.h>
_ismbclower_l	<mbstring.h>
_ismbcupper	<mbstring.h>
_ismbcupper_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
Locale
Interpretation of Multibyte-Character Sequences
is, isw Routines
_ismbb Routines

_ismbslead, _ismbstrail, _ismbslead_l, _ismbstrail_l

10/31/2018 • 2 minutes to read • Edit Online

Performs context-sensitive tests for multibyte-character-string lead bytes and trail bytes and determines whether a given substring pointer points to a lead byte or a trail byte.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ismbslead(
   const unsigned char *str,
   const unsigned char *current
);
int _ismbstrail(
   const unsigned char *str,
   const unsigned char *current
);
int _ismbslead_l(
   const unsigned char *str,
   const unsigned char *current,
   _locale_t locale
);
int _ismbstrail_l(
   const unsigned char *str,
   const unsigned char *current,
   _locale_t locale
);
```

Parameters

str
Pointer to the start of the string or the previous known lead byte.

current
Pointer to the position in the string to be tested.

locale
The locale to use.

Return Value

_ismbslead returns -1 if the character is a lead byte and **_ismbstrail** returns -1 if the character is a trail byte. If the input strings are valid but are not a lead byte or trail byte, these functions return zero. If either argument is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **NULL** and set **errno** to **EINVAL**.

Remarks

_ismbslead and **_ismbstrail** are slower than the **_ismbblead** and **_ismbbtrail** versions because they take the

string context into account.

The versions of these functions that have the **_l** suffix are identical except that for their locale-dependent behavior they use the locale that's passed in instead of the current locale. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_ismbslead	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>
_ismbstrail	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>
_ismbslead_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>
_ismbstrail_l	<mbctype.h> or <mbstring.h>	<ctype.h>,* <limits.h>, <stdlib.h>

* For manifest constants for the test conditions.

For more compatibility information, see Compatibility.

See also

Character Classification
_ismbc Routines
is, isw Routines
_ismbb Routines

isnan, _isnan, _isnanf

2/4/2019 • 2 minutes to read • Edit Online

Tests if a floating-point value is not a number (NAN).

Syntax

```
int isnan(
   /* floating-point */ x
); /* C-only macro */

int _isnan(
   double x
);

int _isnanf(
   float x
); /* x64 only */

template <class T>
bool isnan(
   T x
) throw(); /* C++ only */
```

Parameters

x
The floating-point value to test.

Return Value

In C, the **isnan** macro and the **_isnan** and **_isnanf** functions return a non-zero value if the argument *x* is a NAN; otherwise they return 0.

In C++, the **isnan** template function returns **true** if the argument *x* is a NaN; otherwise it returns **false**.

Remarks

Because a NaN value does not compare as equal to any other NaN value, you must use one of these functions or macros to detect one. A NaN is generated when the result of a floating-point operation can't be represented in IEEE-754 floating-point format for the specified type. For information about how a NaN is represented for output, see printf.

When compiled as C++, the **isnan** macro is not defined, and an **isnan** template function is defined instead. It behaves the same way as the macro, but returns a value of type **bool** instead of an integer.

The **_isnan** and **_isnanf** functions are Microsoft-specific. The **_isnanf** function is only available when compiled for x64.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
isnan, **_isnanf**	<math.h>	<math.h> or <cmath>
_isnan	<float.h>	<float.h> or <cfloat>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
fpclassify
_fpclass, _fpclassf
isfinite, _finite, _finitef
isinf
isnormal

isnormal

4/22/2019 • 2 minutes to read • Edit Online

Determines whether a floating-point value is a normal value.

Syntax

```
int isnormal(
   /* floating-point */ x
); /* C-only macro */

template <class FloatingType>
inline bool isnormal(
   FloatingType x
) throw(); /* C++-only function template */
```

Parameters

x
The floating-point value to test.

Return value

isnormal returns a nonzero value (**true** in C++ code) if the argument *x* is neither zero, subnormal, infinite, nor a NaN. Otherwise, **isnormal** returns 0 (**false** in C++ code).

Remarks

isnormal is a macro when compiled as C, and an inline function template when compiled as C++.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
isnormal	<math.h>	<math.h> or <cmath>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
isfinite, _finite, _finitef
isinf
isnan, _isnan, _isnanf
_fpclass, _fpclassf

ispunct, iswpunct, _ispunct_l, _iswpunct_l

10/31/2018 • 2 minutes to read • Edit Online

Determines whether an integer represents a punctuation character.

Syntax

```
int ispunct(
   int c
);
int iswpunct(
   wint_t c
);
int _ispunct_l(
   int c,
   _locale_t locale
);
int _iswpunct_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a punctuation character. **ispunct** returns a nonzero value for any printable character that is not a space character or a character for which **isalnum** is nonzero. **iswpunct** returns a nonzero value for any printable wide character that is neither the space wide character nor a wide character for which **iswalnum** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

The result of the test condition for the **ispunct** function depends on the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions that do not have the **_l** suffix use the current locale for any locale-dependent behavior; the versions that do have the **_l** suffix are identical except that they use the locale that's passed in instead. For more information, see Locale.

The behavior of **ispunct** and **_ispunct_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ istpunct	**ispunct**	_ismbcpunct	**iswpunct**

Requirements

ROUTINE	REQUIRED HEADER
ispunct	<ctype.h>
iswpunct	<ctype.h> or <wchar.h>
_ispunct_l	<ctype.h>
_iswpunct_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isprint, iswprint, _isprint_l, _iswprint_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents a printable character.

Syntax

```
int isprint(
   int c
);
int iswprint(
   wint_t c
);
int _isprint_l(
   int c,
   _locale_t locale
);
int _iswprint_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
The locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a printable character. **isprint** returns a nonzero value if *c* is a printable character—this includes the space character (0x20 - 0x7E). **iswprint** returns a nonzero value if *c* is a printable wide character—this includes the space wide character. Each of these routines returns 0 if *c* does not satisfy the test condition.

The result of the test condition for these functions depends on the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions that do not have the **_l** suffix use the current locale for any locale-dependent behavior; the versions that do have the **_l** suffix are identical except that they use the locale that's passed in instead. For more information, see Locale.

The behavior of **isprint** and **_isprint_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ istprint	**isprint**	_ismbcprint	**iswprint**

Requirements

ROUTINE	REQUIRED HEADER
isprint	<ctype.h>
iswprint	<ctype.h> or <wchar.h>
_isprint_l	<ctype.h>
_iswprint_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isspace, iswspace, _isspace_l, _iswspace_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents a space character.

Syntax

```
int isspace(
   int c
);
int iswspace(
   wint_t c
);
int _isspace_l(
   int c,
   _locale_t locale
);
int _iswspace_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a space character. **isspace** returns a nonzero value if *c* is a white-space character (0x09 - 0x0D or 0x20). The result of the test condition for the **isspace** function depends on the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions that do not have the **_l** suffix use the current locale for any locale-dependent behavior; the versions that do have the **_l** suffix are identical except that they use the locale that's passed in instead. For more information, see Locale.

iswspace returns a nonzero value if *c* is a wide character that corresponds to a standard white-space character.

The behavior of **isspace** and **_isspace_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ istspace	**isspace**	_ismbcspace	**iswspace**

Requirements

ROUTINE	REQUIRED HEADER
isspace	<ctype.h>
iswspace	<ctype.h> or <wchar.h>
_isspace_l	<ctype.h>
_iswspace_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isupper, _isupper_l, iswupper, _iswupper_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents an uppercase character.

Syntax

```
int isupper(
   int c
);
int _isupper_l (
   int c,
   _locale_t locale
);
int iswupper(
   wint_t c
);
int _iswupper_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of an uppercase letter. **isupper** returns a nonzero value if *c* is an uppercase character (A - Z). **iswupper** returns a nonzero value if *c* is a wide character that corresponds to an uppercase letter, or if *c* is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is nonzero. Each of these routines returns 0 if *c* does not satisfy the test condition.

The versions of these functions that have the **_l** suffix use the locale that's passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **isupper** and **_isupper_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istupper	**isupper**	_ismbcupper	**iswupper**
_istupper_l	**_isupper_l**	_ismbclower, _ismbclower_l, _ismbcupper, _ismbcupper_l	**_iswupper_l**

Requirements

ROUTINE	REQUIRED HEADER
isupper	<ctype.h>
_isupper_l	<ctype.h>
iswupper	<ctype.h> or <wchar.h>
_iswupper_l	<ctype.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

isxdigit, iswxdigit, _isxdigit_l, _iswxdigit_l

3/1/2019 • 2 minutes to read • Edit Online

Determines whether an integer represents a character that is a hexadecimal digit.

Syntax

```
int isxdigit(
   int c
);
int iswxdigit(
   wint_t c
);
int _isxdigit_l(
   int c,
   _locale_t locale
);
int _iswxdigit_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Integer to test.

locale
Locale to use.

Return Value

Each of these routines returns nonzero if *c* is a particular representation of a hexadecimal digit. **isxdigit** returns a nonzero value if *c* is a hexadecimal digit (A - F, a - f, or 0 - 9). **iswxdigit** returns a nonzero value if *c* is a wide character that corresponds to a hexadecimal digit character. Each of these routines returns 0 if *c* does not satisfy the test condition.

For the "C" locale, the **iswxdigit** function does not support Unicode full-width hexadecimal characters.

The versions of these functions that have the **_l** suffix use the locale that's passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

The behavior of **isxdigit** and **_isxdigit_l** is undefined if *c* is not EOF or in the range 0 through 0xFF, inclusive. When a debug CRT library is used and *c* is not one of these values, the functions raise an assertion.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_istxdigit	**isxdigit**	**isxdigit**	**iswxdigit**

Requirements

ROUTINE	REQUIRED HEADER
isxdigit	<ctype.h>
iswxdigit	<ctype.h> or <wchar.h>
_isxdigit_l	<ctype.h>
_iswxdigit_l	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Character Classification
Locale
is, isw Routines

itoa, _itoa, ltoa, _ltoa, ultoa, _ultoa, _i64toa, _ui64toa, _itow, _ltow, _ultow, _i64tow, _ui64tow

3/1/2019 • 7 minutes to read • Edit Online

Converts an integer to a string. More secure versions of these functions are available; see _itoa_s, _itow_s functions.

Syntax

```
char * _itoa( int value, char *buffer, int radix );
char * _ltoa( long value, char *buffer, int radix );
char * _ultoa( unsigned long value, char *buffer, int radix );
char * _i64toa( long long value, char *buffer, int radix );
char * _ui64toa( unsigned long long value, char *buffer, int radix );

wchar_t * _itow( int value, wchar_t *buffer, int radix );
wchar_t * _ltow( long value, wchar_t *buffer, int radix );
wchar_t * _ultow( unsigned long value, wchar_t *buffer, int radix );
wchar_t * _i64tow( long long value, wchar_t *buffer, int radix );
wchar_t * _ui64tow( unsigned long long value, wchar_t *buffer, int radix );

// These Posix versions of the functions have deprecated names:
char * itoa( int value, char *buffer, int radix );
char * ltoa( long value, char *buffer, int radix );
char * ultoa( unsigned long value, char *buffer, int radix );

// The following template functions are C++ only:
template <size_t size>
char *_itoa( int value, char (&buffer)[size], int radix );

template <size_t size>
char *_itoa( long value, char (&buffer)[size], int radix );

template <size_t size>
char *_itoa( unsigned long value, char (&buffer)[size], int radix );

template <size_t size>
char *_i64toa( long long value, char (&buffer)[size], int radix );

template <size_t size>
char * _ui64toa( unsigned long long value, char (&buffer)[size], int radix );

template <size_t size>
wchar_t * _itow( int value, wchar_t (&buffer)[size], int radix );

template <size_t size>
wchar_t * _ltow( long value, wchar_t (&buffer)[size], int radix );

template <size_t size>
wchar_t * _ultow( unsigned long value, wchar_t (&buffer)[size], int radix );

template <size_t size>
wchar_t * _i64tow( long long value, wchar_t (&buffer)[size], int radix );

template <size_t size>
wchar_t * _ui64tow( unsigned long long value, wchar_t (&buffer)[size],
   int radix );
```

Parameters

value
Number to be converted.

buffer
Buffer that holds the result of the conversion.

radix
The base to use for the conversion of *value*, which must be in the range 2-36.

size
Length of the buffer in units of the character type. This parameter is inferred from the *buffer* argument in C++.

Return Value

Each of these functions returns a pointer to *buffer*. There is no error return.

Remarks

The **_itoa**, **_ltoa**, **_ultoa**, **_i64toa**, and **_ui64toa** functions convert the digits of the given *value* argument to a null-terminated character string and store the result (up to 33 characters for **_itoa**, **_ltoa**, and **_ultoa**, and 65 for **_i64toa** and **_ui64toa**) in *buffer*. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (**-**). The **_itow**, **_ltow**, **_ultow**, **_i64tow**, and **_ui64tow** functions are wide-character versions of **_itoa**, **_ltoa**, **_ultoa**, **_i64toa**, and **_ui64toa**, respectively.

> **IMPORTANT**
> These functions can write past the end of a buffer that is too small. To prevent buffer overruns, ensure that *buffer* is large enough to hold the converted digits plus the trailing null-character and a sign character. Misuse of these functions can cause serious security issues in your code.

Because of their potential for security issues, by default, these functions cause deprecation warning C4996: **This function or variable may be unsafe. Consider using** *safe_function* **instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS.** We recommend you change your source code to use the *safe_function* suggested by the warning message. The more secure functions do not write more characters than the specified buffer size. For more information, see _itoa_s, _itow_s functions.

To use these functions without the deprecation warning, define the **_CRT_SECURE_NO_WARNINGS** preprocessor macro before including any CRT headers. You can do this on the command line in a developer command prompt by adding the **/D_CRT_SECURE_NO_WARNINGS** compiler option to the **cl** command. Otherwise, define the macro in your source files. If you use precompiled headers, define the macro at the top of the precompiled header include file, typically stdafx.h. To define the macro in your source code, use a **#define** directive before you include any CRT header, as in this example:

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdlib.h>
```

In C++, these functions have template overloads that invoke their safer counterparts. For more information, see Secure Template Overloads.

The Posix names **itoa**, **ltoa**, and **ultoa** exist as aliases for the **_itoa**, **_ltoa**, and **_ultoa** functions. The Posix names are deprecated because they do not follow the implementation-specific function name conventions of ISO C. By default, these functions cause deprecation warning C4996: **The POSIX name for this item is deprecated. Instead, use the ISO C and C++ conformant name:** *new_name*. We recommend you change

your source code to use the safer versions of these functions, **_itoa_s**, **_ltoa_s**, or **_ultoa_s**. For more information, see _itoa_s, _itow_s functions.

For source code portability, you may prefer to retain the Posix names in your code. To use these functions without the deprecation warning, define both the **_CRT_NONSTDC_NO_WARNINGS** and **_CRT_SECURE_NO_WARNINGS** preprocessor macros before including any CRT headers. You can do this on the command line in a developer command prompt by adding the **/D_CRT_SECURE_NO_WARNINGS** and **/D_CRT_NONSTDC_NO_WARNINGS** compiler options to the **cl** command. Otherwise, define the macros in your source files. If you use precompiled headers, define the macros at the top of the precompiled header include file, typically stdafx.h. To define the macros in your source code, use **#define** directives before you include any CRT header, as in this example:

```
#define _CRT_NONSTDC_NO_WARNINGS 1
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdlib.h>
```

Maximum conversion count macros

To help you create secure buffers for conversions, the CRT includes some convenient macros. These define the size of the buffer required to convert the longest possible value of each integer type, including the null terminator and sign character, for several common bases. To ensure that your conversion buffer is large enough to receive any conversion in the base specified by *radix*, use one of these defined macros when you allocate the buffer. This helps to prevent buffer overrun errors when you convert integral types to strings. These macros are defined when you include either stdlib.h or wchar.h in your source.

To use one of these macros in a string conversion function, declare your conversion buffer of the appropriate character type and use the macro value for the integer type and base as the buffer dimension. This table lists the macros that are appropriate for each function for the listed bases:

Functions	radix	Macros
_itoa, **_itow**	16 10 8 2	**_MAX_ITOSTR_BASE16_COUNT** **_MAX_ITOSTR_BASE10_COUNT** **_MAX_ITOSTR_BASE8_COUNT** **_MAX_ITOSTR_BASE2_COUNT**
_ltoa, **_ltow**	16 10 8 2	**_MAX_LTOSTR_BASE16_COUNT** **_MAX_LTOSTR_BASE10_COUNT** **_MAX_LTOSTR_BASE8_COUNT** **_MAX_LTOSTR_BASE2_COUNT**
_ultoa, **_ultow**	16 10 8 2	**_MAX_ULTOSTR_BASE16_COUNT** **_MAX_ULTOSTR_BASE10_COUNT** **_MAX_ULTOSTR_BASE8_COUNT** **_MAX_ULTOSTR_BASE2_COUNT**
_i64toa, **_i64tow**	16 10 8 2	**_MAX_I64TOSTR_BASE16_COUNT** **_MAX_I64TOSTR_BASE10_COUNT** **_MAX_I64TOSTR_BASE8_COUNT** **_MAX_I64TOSTR_BASE2_COUNT**
_ui64toa, **_ui64tow**	16 10 8 2	**_MAX_U64TOSTR_BASE16_COUNT** **_MAX_U64TOSTR_BASE10_COUNT** **_MAX_U64TOSTR_BASE8_COUNT** **_MAX_U64TOSTR_BASE2_COUNT**

This example uses a conversion count macro to define a buffer large enough to contain an **unsigned long long** in base 2:

```
#include <wchar.h>
#include <iostream>
int main()
{
    wchar_t buffer[_MAX_U64TOSTR_BASE2_COUNT];
    std:wcout << _ui64tow(0xFFFFFFFFFFFFFFFFull, buffer, 2) << std::endl;
}
```

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_itot	**_itoa**	**_itoa**	**_itow**
_ltot	**_ltoa**	**_ltoa**	**_ltow**
_ultot	**_ultoa**	**_ultoa**	**_ultow**
_i64tot	**_i64toa**	**_i64toa**	**_i64tow**
_ui64tot	**_ui64toa**	**_ui64toa**	**_ui64tow**

Requirements

ROUTINE	REQUIRED HEADER
itoa, **ltoa**, **ultoa**	<stdlib.h>
_itoa, **_ltoa**, **_ultoa**, **_i64toa**, **_ui64toa**	<stdlib.h>
_itow, **_ltow**, **_ultow**, **_i64tow**, **_ui64tow**	<stdlib.h> or <wchar.h>

These functions and macros are Microsoft-specific. For more compatibility information, see Compatibility.

Example

This sample demonstrates the use of some of the integer conversion functions. Note the use of the **_CRT_SECURE_NO_WARNINGS** macro to silence warning C4996.

```
// crt_itoa.c
// Compile by using: cl /W4 crt_itoa.c
// This program makes use of the _itoa functions
// in various examples.

#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>      // for printf
#include <string.h>     // for strnlen
#include <stdlib.h>     // for _countof, _itoa fns, _MAX_COUNT macros

int main(void)
{
    char buffer[_MAX_U64TOSTR_BASE2_COUNT];
    int r;

    for (r = 10; r >= 2; --r)
    {
        _itoa(-1, buffer, r);
        printf("base %d: %s (%d chars)\n", r, buffer,
            strnlen(buffer, _countof(buffer)));
    }
    printf("\n");

    for (r = 10; r >= 2; --r)
    {
        _i64toa(-1LL, buffer, r);
        printf("base %d: %s (%d chars)\n", r, buffer,
            strnlen(buffer, _countof(buffer)));
    }
    printf("\n");

    for (r = 10; r >= 2; --r)
    {
        _ui64toa(0xffffffffffffffffULL, buffer, r);
        printf("base %d: %s (%d chars)\n", r, buffer,
            strnlen(buffer, _countof(buffer)));
    }
}
```

```
base 10: -1 (2 chars)
base 9: 12068657453 (11 chars)
base 8: 37777777777 (11 chars)
base 7: 211301422353 (12 chars)
base 6: 1550104015503 (13 chars)
base 5: 32244002423140 (14 chars)
base 4: 3333333333333333 (16 chars)
base 3: 102002022201221111210 (21 chars)
base 2: 11111111111111111111111111111111 (32 chars)

base 10: -1 (2 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777 (22 chars)
base 7: 45012021522523134134601 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 221422030311440042412122430 (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 11112220022122120101211020120210210211220 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)

base 10: 18446744073709551615 (20 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777 (22 chars)
base 7: 45012021522523134134601 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 221422030311440042412122430 (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 11112220022122120101211020120210210211220 (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)
```

See also

Data Conversion

_itoa_s, _itow_s functions

_itoa_s, _ltoa_s, _ultoa_s, _i64toa_s, _ui64toa_s, _itow_s, _ltow_s, _ultow_s, _i64tow_s, _ui64tow_s

3/1/2019 • 4 minutes to read • Edit Online

Converts an integer to a string. These are versions of the _itoa, _itow functions with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _itoa_s( int value, char * buffer, size_t size, int radix );
errno_t _ltoa_s( long value, char * buffer, size_t size, int radix );
errno_t _ultoa_s( unsigned long value, char * buffer, size_t size, int radix );
errno_t _i64toa_s( long long value, char *buffer,
   size_t size, int radix );
errno_t _ui64toa_s( unsigned long long value, char *buffer,
   size_t size, int radix );

errno_t _itow_s( int value, wchar_t *buffer,
   size_t size, int radix );
errno_t _ltow_s( long value, wchar_t *buffer,
   size_t size, int radix );
errno_t _ultow_s( unsigned long value, wchar_t *buffer,
   size_t size, int radix );
errno_t _i64tow_s( long long value, wchar_t *buffer,
   size_t size, int radix );
errno_t _ui64tow_s( unsigned long long value, wchar_t *buffer,
   size_t size, int radix
);

// These template functions are C++ only:
template <size_t size>
errno_t _itoa_s( int value, char (&buffer)[size], int radix );

template <size_t size>
errno_t _ltoa_s( long value, char (&buffer)[size], int radix );

template <size_t size>
errno_t _ultoa_s( unsigned long value, char (&buffer)[size], int radix );

template <size_t size>
errno_t _itow_s( int value, wchar_t (&buffer)[size], int radix );

template <size_t size>
errno_t _ltow_s( long value, wchar_t (&buffer)[size], int radix );

template <size_t size>
errno_t _ultow_s( unsigned long value, wchar_t (&buffer)[size], int radix );
```

Parameters

value
Number to be converted.

buffer
Output buffer that holds the result of the conversion.

size

Size of *buffer* in characters or wide characters.

radix
The radix or numeric base to use to convert *value*, which must be in the range 2-36.

Return value

Zero if successful; an error code on failure. If any of the following conditions applies, the function invokes an invalid parameter handler, as described in Parameter Validation.

Error conditions

VALUE	BUFFER	SIZE	RADIX	RETURN
any	**NULL**	any	any	**EINVAL**
any	any	<=0	any	**EINVAL**
any	any	<= length of the result string required	any	**EINVAL**
any	any	any	*radix* < 2 or *radix* > 36	**EINVAL**

Security issues

These functions can generate an access violation if *buffer* does not point to valid memory and is not **NULL**, or if the length of the buffer is not long enough to hold the result string.

Remarks

Except for the parameters and return value, the **_itoa_s** and **_itow_s** function families have the same behavior as the corresponding less secure **_itoa** and **_itow** versions.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug library versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

The CRT includes convenient macros to define the size of the buffer required to convert the longest possible value of each integer type, including the null terminator and sign character, for several common bases. For information, see Maximum conversion count macros.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_itot_s	**_itoa_s**	**_itoa_s**	**_itow_s**
_ltot_s	**_ltoa_s**	**_ltoa_s**	**_ltow_s**
_ultot_s	**_ultoa_s**	**_ultoa_s**	**_ultow_s**
_i64tot_s	**_i64toa_s**	**_i64toa_s**	**_i64tow_s**

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ui64tot_s	**_ui64toa_s**	**_ui64toa_s**	**_ui64tow_s**

Requirements

ROUTINE	REQUIRED HEADER
_itoa_s, **_ltoa_s**, **_ultoa_s**, **_i64toa_s**, **_ui64toa_s**	<stdlib.h>
_itow_s, **_ltow_s**, **_ultow_s**, **_i64tow_s**, **_ui64tow_s**	<stdlib.h> or <wchar.h>

These functions are Microsoft-specific. For more compatibility information, see Compatibility.

Example

This sample demonstrates the use of a few of the integer conversion functions. Note that the _countof macro only works to determine buffer size when the array declaration is visible to the compiler, and not for parameters that have decayed to pointers.

```
// crt_itoa_s.c
// Compile by using: cl /W4 crt_itoa_s.c
#include <stdlib.h>     // for _itoa_s functions, _countof, count macro
#include <stdio.h>      // for printf
#include <string.h>     // for strnlen

int main( void )
{
    char buffer[_MAX_U64TOSTR_BASE2_COUNT];
    int r;
    for ( r = 10; r >= 2; --r )
    {
        _itoa_s( -1, buffer, _countof(buffer), r );
        printf( "base %d: %s (%d chars)\n",
            r, buffer, strnlen(buffer, _countof(buffer)) );
    }
    printf( "\n" );
    for ( r = 10; r >= 2; --r )
    {
        _i64toa_s( -1LL, buffer, _countof(buffer), r );
        printf( "base %d: %s (%d chars)\n",
            r, buffer, strnlen(buffer, _countof(buffer)) );
    }
    printf( "\n" );
    for ( r = 10; r >= 2; --r )
    {
        _ui64toa_s( 0xffffffffffffffffULL, buffer, _countof(buffer), r );
        printf( "base %d: %s (%d chars)\n",
            r, buffer, strnlen(buffer, _countof(buffer)) );
    }
}
```

```
base 10: -1 (2 chars)
base 9: 12068657453 (11 chars)
base 8: 37777777777 (11 chars)
base 7: 211301422353 (12 chars)
base 6: 1550104015503 (13 chars)
base 5: 32244002423140 (14 chars)
base 4: 3333333333333333 (16 chars)
base 3: 102002022201221111210 (21 chars)
base 2: 11111111111111111111111111111111 (32 chars)

base 10: -1 (2 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 450120215225231341346O1 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 221422030311440042412112243O (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 1111222002212212010121102012021021022O (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)

base 10: 18446744073709551615 (20 chars)
base 9: 145808576354216723756 (21 chars)
base 8: 1777777777777777777777 (22 chars)
base 7: 450120215225231341346O1 (23 chars)
base 6: 3520522010102100444244423 (25 chars)
base 5: 221422030311440042412112243O (28 chars)
base 4: 33333333333333333333333333333333 (32 chars)
base 3: 1111222002212212010121102012021021022O (41 chars)
base 2: 1111111111111111111111111111111111111111111111111111111111111111 (64 chars)
```

See also

Data Conversion
_itoa, _itow functions

j0, j1, jn

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _j0, _j1, _jn instead.

kbhit

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _kbhit instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_kbhit

10/31/2018 • 2 minutes to read • Edit Online

Checks the console for keyboard input.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _kbhit( void );
```

Return Value

_kbhit returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Remarks

The **_kbhit** function checks the console for a recent keystroke. If the function returns a nonzero value, a keystroke is waiting in the buffer. The program can then call **_getch** or **_getche** to get the keystroke.

Requirements

ROUTINE	REQUIRED HEADER
_kbhit	<conio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_kbhit.c
// compile with: /c
/* This program loops until the user
* presses a key. If _kbhit returns nonzero, a
* keystroke is waiting in the buffer. The program
* can call _getch or _getche to get the keystroke.
*/

#include <conio.h>
#include <stdio.h>

int main( void )
{
   /* Display message until key is pressed. */
   while( !_kbhit() )
      _cputs( "Hit me!! " );

   /* Use _getch to throw key away. */
   printf( "\nKey struck was '%c'\n", _getch() );
}
```

Sample Output

```
Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!! Hit me!!
Key struck was 'q'
```

See also

Console and Port I/O

ldexp, ldexpf, ldexpl

10/31/2018 • 2 minutes to read • Edit Online

Multiplies a floating-point number by an integral power of two.

Syntax

```
double ldexp(
   double x,
   int exp
);
float ldexp(
   float x,
   int exp
);  // C++ only
long double ldexp(
   long double x,
   int exp
);  // C++ only
float ldexpf(
   float x,
   int exp
);
long double ldexpl(
   long double x,
   int exp
);
```

Parameters

x
Floating-point value.

exp
Integer exponent.

Return Value

The **ldexp** functions return the value of $x * 2^{exp}$ if successful. On overflow, and depending on the sign of *x*, **ldexp** returns +/- **HUGE_VAL**; the **errno** value is set to **ERANGE**.

For more information about **errno** and possible error return values, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Because C++ allows overloading, you can call overloads of **ldexp** that take **float** or **long double** types. In a C program, **ldexp** always takes a **double** and an **int** and returns a **double**.

Requirements

ROUTINE	C HEADER	C++ HEADER
ldexp, **ldexpf**, **ldexpl**	<math.h>	<cmath>

For compatibility information, see Compatibility.

Example

```
// crt_ldexp.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 4.0, y;
   int p = 3;

   y = ldexp( x, p );
   printf( "%2.1f times two to the power of %d is %2.1f\n", x, p, y );
}
```

Output

```
4.0 times two to the power of 3 is 32.0
```

See also

Floating-Point Support
frexp
modf, modff, modfl

2 minutes to read

lfind

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _lfind or security-enhanced _lfind_s instead.

_lfind

10/31/2018 • 2 minutes to read • Edit Online

Performs a linear search for the specified key. A more secure version of this function is available; see _lfind_s.

Syntax

```
void *_lfind(
   const void *key,
   const void *base,
   unsigned int *num,
   unsigned int width,
   int (__cdecl *compare)(const void *, const void *)
);
```

Parameters

key
Object to search for.

base
Pointer to the base of search data.

number
Number of array elements.

width
Width of array elements.

compare
Pointer to comparison routine. The first parameter is a pointer to key for search. The second parameter is a pointer to array element to be compared with key.

Return Value

If the key is found, **_lfind** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lfind** returns **NULL**.

Remarks

The **_lfind** function performs a linear search for the value *key* in an array of *number* elements, each of *width* bytes. Unlike **bsearch**, **_lfind** does not require the array to be sorted. The *base* argument is a pointer to the base of the array to be searched. The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. **_lfind** calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The *compare* routine must compare the elements and then return nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

This function validates its parameters. If *compare*, *key* or *number* is **NULL**, or if *base* is **NULL** and *number* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_lfind	<search.h>

For more compatibility information, see Compatibility.

Example

```
// crt_lfind.c
// This program uses _lfind to search a string array
// for an occurrence of "hello".

#include <search.h>
#include <string.h>
#include <stdio.h>

int compare(const void *arg1, const void *arg2 )
{
   return( _stricmp( * (char**)arg1, * (char**)arg2 ) );
}

int main( )
{
   char *arr[] = {"Hi", "Hello", "Bye"};
   int n = sizeof(arr) / sizeof(char*);
   char **result;
   char *key = "hello";

   result = (char **)_lfind( &key, arr,
                     &n, sizeof(char *), compare );

   if( result )
      printf( "%s found\n", *result );
   else
      printf( "hello not found!\n" );
}
```

```
Hello found
```

See also

Searching and Sorting
_lfind_s
bsearch
_lsearch
qsort

_lfind_s

10/31/2018 • 3 minutes to read • Edit Online

Performs a linear search for the specified key. A version of _lfind with security enhancements as described in Security Features in the CRT.

Syntax

```
void *_lfind_s(
   const void *key,
   const void *base,
   unsigned int *num,
   size_t size,
   int (__cdecl *compare)(void *, const void *, const void *),
   void * context
);
```

Parameters

key
Object to search for.

base
Pointer to the base of search data.

number
Number of array elements.

size
Size of array elements in bytes.

compare
Pointer to comparison routine. The first parameter is the *context* pointer. The second parameter is a pointer to key for search. The third parameter is a pointer to array element to be compared with key.

context
A pointer to an object that might be accessed in the comparison function.

Return Value

If the key is found, **_lfind_s** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lfind_s** returns **NULL**.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Error Conditions

KEY	BASE	COMPARE	NUM	SIZE	ERRNO
NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**

KEY	BASE	COMPARE	NUM	SIZE	ERRNO
any	any	any	any	zero	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **_lfind_s** function performs a linear search for the value *key* in an array of *number* elements, each of *width* bytes. Unlike **bsearch_s**, **_lfind_s** does not require the array to be sorted. The *base* argument is a pointer to the base of the array to be searched. The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. **_lfind_s** calls the *compare* routine one or more times during the search, passing the *context* pointer and pointers to two array elements on each call. The *compare* routine must compare the elements then return nonzero (meaning that the elements are different) or 0 (meaning the elements are identical).

_lfind_s is similar to **_lfind** except for the addition of the *context* pointer to the arguments of the comparison function and the parameter list of the function. The *context* pointer can be useful if the searched data structure is part of an object and the *compare* function needs to access members of the object. The *compare* function can cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* parameter makes **_lfind_s** more secure because additional context can be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

ROUTINE	REQUIRED HEADER
_lfind_s	<search.h>

For more compatibility information, see Compatibility.

Example

```
// crt_lfind_s.cpp
// This program uses _lfind_s to search a string array,
// passing a locale as the context.
// compile with: /EHsc
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;

// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
// Codepage 850 is the OEM codepage used by the command line,
// so \x00e1 is the German Sharp S

char *array1[] = { "wei\x00e1", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };
```

```
                    weit" };

#define GERMAN_LOCALE "German_Germany.850"

#endif

#ifdef CODEPAGE_1252
   // If using codepage 1252 (ISO 8859-1, Latin-1), use \x00df
   // for the German Sharp S
char *array1[] = { "wei\x00df", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };

#define GERMAN_LOCALE "German_Germany.1252"

#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making it vulnerable to thread conflicts
// (if this were a multithreaded program).

int compare( void *pvlocale, const void *str1, const void *str2)
{
    char *s1 = *(char**)str1;
    char *s2 = *(char**)str2;

    locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

    return use_facet< collate<char> >(loc).compare(
       s1, s1+strlen(s1),
       s2, s2+strlen(s2) );
}

void find_it( char *key, char *array[], unsigned int num, locale &loc )
{
   char **result = (char **)_lfind_s( &key, array,
                      &num, sizeof(char *), compare, &loc );
   if( result )
      printf( "%s found\n", *result );
   else
      printf( "%s not found\n", key );
}

int main( )
{
   find_it( "weit", array1, sizeof(array1)/sizeof(char*), locale(GERMAN_LOCALE) );
}
```

```
weit found
```

See also

Searching and Sorting
bsearch_s
_lsearch_s
qsort_s
_lfind

lgamma, lgammaf, lgammal

10/31/2018 • 2 minutes to read • Edit Online

Determines the natural logarithm of the absolute value of the gamma function of the specified value.

Syntax

```
double lgamma( double x );
float lgammaf( float x );
long double lgammal( long double x );
```

```
float lgamma( float x ); //C++ only
long double lgamma( long double x ); //C++ only
```

Parameters

x
The value to compute.

Return Value

If successful, return the natural logarithm of the absolute value of the gamma function of *x*.

ISSUE	RETURN
x = NaN	NaN
x = ±0	+INFINITY
x= negative integer	+INFINITY
±INFINITY	+INFINITY
pole error	+HUGE_VAL, +HUGE_VALF, or +HUGE_VALL
overflow range error	±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **lgamma** that take and return **float** and **long double** types. In a C program, **lgamma** always takes and returns a **double**.

If x is a rational number, this function returns the logarithm of the factorial of (x - 1).

Requirements

FUNCTION	C HEADER	C++ HEADER
lgamma, **lgammaf**, **lgammal**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
tgamma, tgammaf, tgammal

localeconv

10/31/2018 • 3 minutes to read • Edit Online

Gets detailed information on locale settings.

Syntax

```
struct lconv *localeconv( void );
```

Return Value

localeconv returns a pointer to a filled-in object of type struct lconv. The values contained in the object are copied from the locale settings in thread-local storage, and can be overwritten by subsequent calls to **localeconv**. Changes made to the values in this object do not modify the locale settings. Calls to setlocale with *category* values of **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** overwrite the contents of the structure.

Remarks

The **localeconv** function gets detailed information about numeric formatting for the current locale. This information is stored in a structure of type **lconv**. The **lconv** structure, defined in LOCALE.H, contains the following members:

FIELD	MEANING
decimal_point, _W_decimal_point	Pointer to decimal-point character for nonmonetary quantities.
thousands_sep, _W_thousands_sep	Pointer to character that separates groups of digits to left of decimal point for nonmonetary quantities.
grouping	Pointer to a **char**-sized integer that contains the size of each group of digits in nonmonetary quantities.
int_curr_symbol, _W_int_curr_symbol	Pointer to international currency symbol for current locale. First three characters specify alphabetic international currency symbol as defined in the *ISO 4217 Codes for the Representation of Currency and Funds* standard. Fourth character (immediately preceding null character) separates international currency symbol from monetary quantity.
currency_symbol, _W_currency_symbol	Pointer to local currency symbol for current locale.
mon_decimal_point, _W_mon_decimal_point	Pointer to decimal-point character for monetary quantities.
mon_thousands_sep, _W_mon_thousands_sep	Pointer to separator for groups of digits to left of decimal place in monetary quantities.

FIELD	MEANING
mon_grouping	Pointer to a **char**-sized integer that contains the size of each group of digits in monetary quantities.
positive_sign, _W_positive_sign	String denoting sign for nonnegative monetary quantities.
negative_sign, _W_negative_sign	String denoting sign for negative monetary quantities.
int_frac_digits	Number of digits to right of decimal point in internationally formatted monetary quantities.
frac_digits	Number of digits to right of decimal point in formatted monetary quantities.
p_cs_precedes	Set to 1 if currency symbol precedes value for nonnegative formatted monetary quantity. Set to 0 if symbol follows value.
p_sep_by_space	Set to 1 if currency symbol is separated by space from value for nonnegative formatted monetary quantity. Set to 0 if there is no space separation.
n_cs_precedes	Set to 1 if currency symbol precedes value for negative formatted monetary quantity. Set to 0 if symbol succeeds value.
n_sep_by_space	Set to 1 if currency symbol is separated by space from value for negative formatted monetary quantity. Set to 0 if there is no space separation.
p_sign_posn	Position of positive sign in nonnegative formatted monetary quantities.
n_sign_posn	Position of positive sign in negative formatted monetary quantities.

Except as specified, members of the **lconv** structure that have `char *` and `wchar_t *` versions are pointers to strings. Any of these that equals **""** (or **L""** for **wchar_t ***) is either of zero length or not supported in the current locale. Note that **decimal_point** and **_W_decimal_point** are always supported and of nonzero length.

The **char** members of the structure are small nonnegative numbers, not characters. Any of these that equals **CHAR_MAX** is not supported in the current locale.

The values of **grouping** and **mon_grouping** are interpreted according to the following rules:

- **CHAR_MAX** - Do not perform any further grouping.
- 0 - Use previous element for each of remaining digits.
- *n* - Number of digits that make up current group. Next element is examined to determine size of next group of digits before current group.

The values for **int_curr_symbol** are interpreted according to the following rules:

- The first three characters specify the alphabetic international currency symbol as defined in the *ISO*

4217 Codes for the Representation of Currency and Funds standard.

- The fourth character (immediately preceding the null character) separates the international currency symbol from the monetary quantity.

The values for **p_cs_precedes** and **n_cs_precedes** are interpreted according to the following rules (the **n_cs_precedes** rule is in parentheses):

- 0 - Currency symbol follows value for nonnegative (negative) formatted monetary value.
- 1 - Currency symbol precedes value for nonnegative (negative) formatted monetary value.

The values for **p_sep_by_space** and **n_sep_by_space** are interpreted according to the following rules (the **n_sep_by_space** rule is in parentheses):

- 0 - Currency symbol is separated from value by space for nonnegative (negative) formatted monetary value.
- 1 - There is no space separation between currency symbol and value for nonnegative (negative) formatted monetary value.

The values for **p_sign_posn** and **n_sign_posn** are interpreted according to the following rules:

- 0 - Parentheses surround quantity and currency symbol.
- 1 - Sign string precedes quantity and currency symbol.
- 2 - Sign string follows quantity and currency symbol.
- 3 - Sign string immediately precedes currency symbol.
- 4 - Sign string immediately follows currency symbol.

Requirements

ROUTINE	REQUIRED HEADER
localeconv	<locale.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Locale
setlocale
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

localtime, _localtime32, _localtime64

10/31/2018 • 3 minutes to read • Edit Online

Converts a time value and corrects for the local time zone. More secure versions of these functions are available; see localtime_s, _localtime32_s, _localtime64_s.

Syntax

```
struct tm *localtime( const time_t *sourceTime );
struct tm *_localtime32( const __time32_t *sourceTime );
struct tm *_localtime64( const __time64_t *sourceTime );
```

Parameters

sourceTime
Pointer to stored time.

Return Value

Return a pointer to the structure result, or **NULL** if the date passed to the function is:

- Before midnight, January 1, 1970.
- After 03:14:07, January 19, 2038, UTC (using **_time32** and **time32_t**).
- After 23:59:59, December 31, 3000, UTC (using **_time64** and **__time64_t**).

_localtime64, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, December 31, 3000, coordinated universal time (UTC), whereas **_localtime32** represents dates through 23:59:59 January 18, 2038, UTC.

localtime is an inline function which evaluates to **_localtime64**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **localtime** to evaluate to **_localtime32**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

The fields of the structure type tm store the following values, each of which is an **int**:

FIELD	DESCRIPTION
tm_sec	Seconds after minute (0 - 59).
tm_min	Minutes after hour (0 - 59).
tm_hour	Hours since midnight (0 - 23).
tm_mday	Day of month (1 - 31).
tm_mon	Month (0 - 11; January = 0).
tm_year	Year (current year minus 1900).

FIELD	DESCRIPTION
tm_wday	Day of week (0 - 6; Sunday = 0).
tm_yday	Day of year (0 - 365; January 1 = 0).
tm_isdst	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown.

If the **TZ** environment variable is set, the C run-time library assumes rules appropriate to the United States for implementing the calculation of daylight-saving time (DST).

Remarks

The **localtime** function converts a time stored as a time_t value and stores the result in a structure of type tm. The **long** value *sourceTime* represents the seconds elapsed since midnight (00:00:00), January 1, 1970, UTC. This value is usually obtained from the time function.

Both the 32-bit and 64-bit versions of gmtime, mktime, mkgmtime, and **localtime** all use a single **tm** structure per thread for the conversion. Each call to one of these routines destroys the result of the previous call.

localtime corrects for the local time zone if the user first sets the global environment variable **TZ**. When **TZ** is set, three other environment variables (**_timezone**, **_daylight**, and **_tzname**) are automatically set as well. If the **TZ** variable is not set, **localtime** attempts to use the time zone information specified in the Date/Time application in Control Panel. If this information cannot be obtained, PST8PDT, which signifies the Pacific Time Zone, is used by default. See _tzset for a description of these variables. **TZ** is a Microsoft extension and not part of the ANSI standard definition of **localtime**.

> **NOTE**
> The target environment should try to determine whether daylight saving time is in effect.

These functions validate their parameters. If *sourceTime* is a null pointer, or if the *sourceTime* value is negative, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **NULL** and set **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
localtime, **_localtime32**, **_localtime64**	<time.h>	<ctime> or <time.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_localtime.cpp
// compile with: /W3
// This program uses _time64 to get the current time
// and then uses localtime64() to convert this time to a structure
// representing the local time. The program converts the result
// from a 24-hour clock to a 12-hour clock and determines the
// proper extension (AM or PM).

#include <stdio.h>
#include <string.h>
#include <time.h>

int main( void )
{
    struct tm *newtime;
    char am_pm[] = "AM";
    __time64_t long_time;

    _time64( &long_time );             // Get time as 64-bit integer.
                                       // Convert to local time.
    newtime = _localtime64( &long_time ); // C4996
    // Note: _localtime64 deprecated; consider _localetime64_s

    if( newtime->tm_hour > 12 )        // Set up extension.
        strcpy_s( am_pm, sizeof(am_pm), "PM" );
    if( newtime->tm_hour > 12 )        // Convert from 24-hour
        newtime->tm_hour -= 12;        //   to 12-hour clock.
    if( newtime->tm_hour == 0 )        // Set hour to 12 if midnight.
        newtime->tm_hour = 12;

    char buff[30];
    asctime_s( buff, sizeof(buff), newtime );
    printf( "%.19s %s\n", buff, am_pm );
}
```

```
Tue Feb 12 10:05:58 AM
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64
_tzset

localtime_s, _localtime32_s, _localtime64_s

10/31/2018 • 4 minutes to read • Edit Online

Converts a **time_t** time value to a **tm** structure, and corrects for the local time zone. These are versions of localtime, _localtime32, _localtime64 with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t localtime_s(
   struct tm* const tmDest,
   time_t const* const sourceTime
);
errno_t _localtime32_s(
   struct tm* tmDest,
   __time32_t const* sourceTime
);
errno_t _localtime64_s(
   struct tm* tmDest,
   __time64_t const* sourceTime
);
```

Parameters

tmDest
Pointer to the time structure to be filled in.

sourceTime
Pointer to the stored time.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in Errno.h. For a listing of these errors, see errno.

Error Conditions

TMDEST	*SOURCETIME*	RETURN VALUE	VALUE IN *TMDEST*	INVOKES INVALID PARAMETER HANDLER
NULL	any	**EINVAL**	Not modified	Yes
Not **NULL** (points to valid memory)	**NULL**	**EINVAL**	All fields set to -1	Yes
Not **NULL** (points to valid memory)	less than 0 or greater than **_MAX__TIME64_T**	**EINVAL**	All fields set to -1	No

In the case of the first two error conditions, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

The **_localtime32_s** function converts a time stored as a time_t value and stores the result in a structure of

type tm. The **long** value *sourceTime* represents the seconds elapsed since midnight (00:00:00), January 1, 1970, UTC. This value is usually obtained from the time function.

_localtime32_s corrects for the local time zone if the user first sets the global environment variable **TZ**. When **TZ** is set, three other environment variables (**_timezone**, **_daylight**, and **_tzname**) are automatically set as well. If the **TZ** variable is not set, **localtime32_s** attempts to use the time zone information specified in the Date/Time application in Control Panel. If this information cannot be obtained, PST8PDT, which signifies the Pacific time zone, is used by default. See _tzset for a description of these variables. **TZ** is a Microsoft extension and not part of the ANSI standard definition of **localtime**.

> **NOTE**
> The target environment should try to determine whether daylight saving time is in effect.

_localtime64_s, which uses the **__time64_t** structure, allows dates to be expressed up through 23:59:59, January 18, 3001, coordinated universal time (UTC), whereas **_localtime32_s** represents dates through 23:59:59 January 18, 2038, UTC.

localtime_s is an inline function which evaluates to **_localtime64_s**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. Doing this will cause **localtime_s** to evaluate to **_localtime32_s**. This is not recommended because your application may fail after January 18, 2038, and it is not allowed on 64-bit platforms.

The fields of the structure type tm store the following values, each of which is an **int**.

FIELD	DESCRIPTION
tm_sec	Seconds after minute (0 - 59).
tm_min	Minutes after hour (0 - 59).
tm_hour	Hours since midnight (0 - 23).
tm_mday	Day of month (1 - 31).
tm_mon	Month (0 - 11; January = 0).
tm_year	Year (current year minus 1900).
tm_wday	Day of week (0 - 6; Sunday = 0).
tm_yday	Day of year (0 - 365; January 1 = 0).
tm_isdst	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown.

If the **TZ** environment variable is set, the C run-time library assumes rules appropriate to the United States for implementing the calculation of daylight saving time (DST).

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
localtime_s, **_localtime32_s**, **_localtime64_s**	<time.h>	<ctime> or <time.h>

For more compatibility information, see Compatibility.

Example

```
// crt_localtime_s.c
// This program uses _time64 to get the current time
// and then uses _localtime64_s() to convert this time to a structure
// representing the local time. The program converts the result
// from a 24-hour clock to a 12-hour clock and determines the
// proper extension (AM or PM).

#include <stdio.h>
#include <string.h>
#include <time.h>

int main( void )
{
    struct tm newtime;
    char am_pm[] = "AM";
    __time64_t long_time;
    char timebuf[26];
    errno_t err;

    // Get time as 64-bit integer.
    _time64( &long_time );
    // Convert to local time.
    err = _localtime64_s( &newtime, &long_time );
    if (err)
    {
        printf("Invalid argument to _localtime64_s.");
        exit(1);
    }
    if( newtime.tm_hour > 12 )        // Set up extension.
        strcpy_s( am_pm, sizeof(am_pm), "PM" );
    if( newtime.tm_hour > 12 )        // Convert from 24-hour
        newtime.tm_hour -= 12;        // to 12-hour clock.
    if( newtime.tm_hour == 0 )        // Set hour to 12 if midnight.
        newtime.tm_hour = 12;

    // Convert to an ASCII representation.
    err = asctime_s(timebuf, 26, &newtime);
    if (err)
    {
        printf("Invalid argument to asctime_s.");
        exit(1);
    }
    printf( "%.19s %s\n", timebuf, am_pm );
}
```

```
Fri Apr 25 01:19:27 PM
```

See also

Time Management
asctime_s, _wasctime_s

ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_ftime, _ftime32, _ftime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
time, _time32, _time64
_tzset

_lock_file

10/31/2018 • 2 minutes to read • Edit Online

Locks a **FILE** object to ensure consistency for threads accessing the **FILE** object concurrently.

Syntax

```
void _lock_file( FILE* file );
```

Parameters

file
File handle.

Remarks

The **_lock_file** function locks the **FILE** object specified by *file*. The underlying file is not locked by **_lock_file**. Use _unlock_file to release the lock on the file. Calls to **_lock_file** and **_unlock_file** must be matched in a thread.

Requirements

ROUTINE	REQUIRED HEADER
_lock_file	<stdio.h>

For more compatibility information, see Compatibility.

Example

```
// crt_lock_file.c
// This example creates multiple threads that write to standard output
// concurrently, first with _file_lock, then without.

#include <stdio.h>
#include <process.h>// _beginthread
#include <windows.h>// HANDLE

void Task_locked( void* str )
{
    for( int i=0; i<1000; ++i )
    {
        _lock_file( stdout );
        for( char* cp = (char*)str; *cp; ++cp )
        {
            _fputc_nolock( *cp, stdout );
        }
        _unlock_file( stdout );
    }
}

void Task_unlocked( void* str )
{
    for( int i=0; i<1000; ++i )
    {
        for( char* cp = (char*)str; *cp; ++cp )
        {
            fputc( *cp, stdout );
        }
    }
}

int main()
{
    HANDLE h[3];
    h[0] = (HANDLE)_beginthread( &Task_locked, 0, "First\n" );
    h[1] = (HANDLE)_beginthread( &Task_locked, 0, "Second\n" );
    h[2] = (HANDLE)_beginthread( &Task_locked, 0, "Third\n" );

    WaitForMultipleObjects( 3, h, true, INFINITE );

    h[0] = (HANDLE)_beginthread( &Task_unlocked, 0, "First\n" );
    h[1] = (HANDLE)_beginthread( &Task_unlocked, 0, "Second\n" );
    h[2] = (HANDLE)_beginthread( &Task_unlocked, 0, "Third\n" );

    WaitForMultipleObjects( 3, h, true, INFINITE );
}
```

```
...
First
Second
First
Second
Third
Second
Third
Second
...
FSiercsotn
dF
iSrescto
nFdi
rSsetc
oFnidr
sSte
cFoinrds
tS
eFciornsdt
```

See also

File Handling
_creat, _wcreat
_open, _wopen
_unlock_file

locking

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _locking instead.

_locking

11/8/2018 • 2 minutes to read • Edit Online

Locks or unlocks bytes of a file.

Syntax

```
int _locking(
   int fd,
   int mode,
   long nbytes
);
```

Parameters

fd
File descriptor.

mode
Locking action to perform.

nbytes
Number of bytes to lock.

Return Value

_locking returns 0 if successful. A return value of -1 indicates failure, in which case errno is set to one of the following values.

ERRNO VALUE	CONDITION
EACCES	Locking violation (file already locked or unlocked).
EBADF	Invalid file descriptor.
EDEADLOCK	Locking violation. Returned when the **_LK_LOCK** or **_LK_RLCK** flag is specified and the file cannot be locked after 10 attempts.
EINVAL	An invalid argument was given to **_locking**.

If the failure is due to a bad parameter, such as an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation.

Remarks

The **_locking** function locks or unlocks *nbytes* bytes of the file specified by *fd*. Locking bytes in a file prevents access to those bytes by other processes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbytes* bytes. It is possible to lock bytes past end of file.

mode must be one of the following manifest constants, which are defined in Locking.h.

MODE VALUE	EFFECT
_LK_LOCK	Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, the constant returns an error.
_LK_NBLCK	Locks the specified bytes. If the bytes cannot be locked, the constant returns an error.
_LK_NBRLCK	Same as **_LK_NBLCK**.
_LK_RLCK	Same as **_LK_LOCK**.
_LK_UNLCK	Unlocks the specified bytes, which must have been previously locked.

Multiple regions of a file that do not overlap can be locked. A region being unlocked must have been previously locked. **_locking** does not merge adjacent regions; if two locked regions are adjacent, each region must be unlocked separately. Regions should be locked only briefly and should be unlocked before closing a file or exiting the program.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_locking	<io.h> and <sys/locking.h>	<errno.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_locking.c
/* This program opens a file with sharing. It locks
* some bytes before reading them, then unlocks them. Note that the
* program works correctly only if the file exists.
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/locking.h>
#include <share.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <io.h>

int main( void )
{
   int  fh, numread;
   char buffer[40];

   /* Quit if can't open file or system doesn't
    * support sharing.
    */
   errno_t err = _sopen_s( &fh, "crt_locking.txt", _O_RDONLY, _SH_DENYNO,
                           _S_IREAD | _S_IWRITE );
   printf( "%d %d\n", err, fh );
   if( err != 0 )
      exit( 1 );

   /* Lock some bytes and read them. Then unlock. */
   if( _locking( fh, LK_NBLCK, 30L ) != -1 )
   {
      long lseek_ret;
      printf( "No one can change these bytes while I'm reading them\n" );
      numread = _read( fh, buffer, 30 );
      buffer[30] = '\0';
      printf( "%d bytes read: %.30s\n", numread, buffer );
      lseek_ret = _lseek( fh, 0L, SEEK_SET );
      _locking( fh, LK_UNLCK, 30L );
      printf( "Now I'm done. Do what you will with them\n" );
   }
   else
      perror( "Locking failed\n" );

   _close( fh );
}
```

Input: crt_locking.txt

```
The first thirty bytes of this file will be locked.
```

Sample Output

```
No one can change these bytes while I'm reading them
30 bytes read: The first thirty bytes of this
Now I'm done. Do what you will with them
```

See also

File Handling

_creat, _wcreat

_open, _wopen

log, logf, logl, log10, log10f, log10l

10/31/2018 • 2 minutes to read • Edit Online

Calculates logarithms.

Syntax

```
double log( double x );
float logf( float x );
long double logl( double x );
double log10( double x );
float log10f ( float x );
long double log10l( double x );
```

```
float log( float x );  // C++ only
long double log( long double x );  // C++ only
float log10( float x );  // C++ only
long double log10( long double x );  // C++ only
```

Parameters

x
Value whose logarithm is to be found.

Return Value

The **log** functions return the natural logarithm (base *e*) of *x* if successful. The **log10** functions return the base-10 logarithm. If *x* is negative, these functions return an indefinite (IND), by default. If *x* is 0, they return infinity (INF).

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN, IND	none	_DOMAIN
± 0	ZERODIVIDE	_SING
x < 0	INVALID	_DOMAIN

log and **log10** have an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

Remarks

C++ allows overloading, so you can call overloads of **log** and **log10** that take and return **float** or **long double** values. In a C program, **log** and **log10** always take and return a **double**.

Requirements

ROUTINE	REQUIRED HEADER
log, **logf**, **logl**, **log10**, **log10f**, **log10l**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_log.c
/* This program uses log and log10
* to calculate the natural logarithm and
* the base-10 logarithm of 9,000.
*/

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 9000.0;
   double y;

   y = log( x );
   printf( "log( %.2f ) = %f\n", x, y );
   y = log10( x );
   printf( "log10( %.2f ) = %f\n", x, y );
}
```

```
log( 9000.00 ) = 9.104980
log10( 9000.00 ) = 3.954243
```

To generate logarithms for other bases, use the mathematical relation: log base b of a == natural log (a) / natural log (b).

```
// logbase.cpp
#include <math.h>
#include <stdio.h>

double logbase(double a, double base)
{
   return log(a) / log(base);
}

int main()
{
   double x = 65536;
   double result;

   result = logbase(x, 2);
   printf("Log base 2 of %lf is %lf\n", x, result);
}
```

```
Log base 2 of 65536.000000 is 16.000000
```

See also

Floating-Point Support

exp, expf, expl
_matherr
pow, powf, powl
_CIlog
_CIlog10

log1p, log1pf, log1pl

11/9/2018 • 2 minutes to read • Edit Online

Computes the natural logarithm of 1 plus the specified value.

Syntax

```
double log1p(
   double x
);

float log1p(
   float x
); //C++ only

long double log1p(
   long double x
); //C++ only

float log1pf(
   float x
);

long double log1pl(
   long double x
);
```

Parameters

x
The floating-point argument.

Return Value

If successful, returns the natural (base-*e*) log of (*x* + 1).

Otherwise, may return one of the following values:

INPUT	RESULT	SEH EXCEPTION	ERRNO
+inf	+inf		
Denormals	Same as input	UNDERFLOW	
±0	Same as input		
-1	-inf	DIVBYZERO	ERANGE
< -1	nan	INVALID	EDOM
-inf	nan	INVALID	EDOM
±SNaN	Same as input	INVALID	

INPUT	RESULT	SEH EXCEPTION	ERRNO
±QNaN, indefinite	Same as input		

The **errno** value is set to ERANGE if *x* = -1. The **errno** value is set to **EDOM** if *x* < -1.

Remarks

The **log1p** functions may be more accurate than using `log(x + 1)` when *x* is near 0.

Because C++ allows overloading, you can call overloads of **log1p** that take and return **float** and **long double** types. In a C program, **log1p** always takes and returns a **double**.

If *x* is a natural number, this function returns the logarithm of the factorial of (*x* - 1).

Requirements

FUNCTION	C HEADER	C++ HEADER
log1p, **log1pf**, **log1pl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
log2, log2f, log2l
log, logf, log10, log10f

log2, log2f, log2l

11/9/2018 • 2 minutes to read • Edit Online

Determines the binary (base-2) logarithm of the specified value.

Syntax

```
double log2(
   double x
);

float log2(
   float x
); //C++ only

long double log2(
   long double x
); //C++ only

float log2f(
   float x
);

long double log2l(
   long double x
);
```

Parameters

x
The value to determine the base-2 logarithm of.

Return Value

On success, returns return log2 *x*.

Otherwise, may return one of the following values:

ISSUE	RETURN
x < 0	NaN
x = ±0	-INFINITY
x = 1	+0
+INFINITY	+INFINITY
NaN	NaN
domain error	NaN
pole error	-HUGE_VAL, -HUGE_VALF, or -HUGE_VALL

Errors are reported as specified in _matherr.

Remarks

If x is an integer, this function essentially returns the zero-based index of the most significant 1 bit of *x*.

Requirements

FUNCTION	C HEADER	C++ HEADER
log2, **log2f**, **log2l**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
exp2, exp2f, exp2l
log, logf, log10, log10f

logb, logbf, logbl, _logb, _logbf

10/31/2018 • 2 minutes to read • Edit Online

Extracts the exponent value of a floating-point argument.

Syntax

```
double logb(
   double x
);
float logb(
   float x
); // C++ only
long double logb(
   long double x
); // C++ only
float logbf(
   float x
);
long double logbl(
   long double x
);
double _logb(
   double x
);
float _logbf(
   float x
);
```

Parameters

x
A floating-point value.

Return Value

logb returns the unbiased exponent value of *x* as a signed integer represented as a floating-point value.

Remarks

The **logb** functions extract the exponential value of the floating-point argument *x*, as though *x* were represented with infinite range. If the argument *x* is denormalized, it is treated as if it were normalized.

Because C++ allows overloading, you can call overloads of **logb** that take and return **float** or **long double** values. In a C program, **logb** always takes and returns a **double**.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	None	_DOMAIN
± 0	ZERODIVIDE	_SING

Requirements

ROUTINE	REQUIRED HEADER
_logb	<float.h>
logb, **logbf**, **logbl**, **_logbf**	<math.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Floating-Point Support
frexp

longjmp

3/1/2019 • 2 minutes to read • Edit Online

Restores the stack environment and execution locale set by a `setjmp` call.

Syntax

```
void longjmp(
   jmp_buf env,
   int value
);
```

Parameters

env
Variable in which environment is stored.

value
Value to be returned to `setjmp` call.

Remarks

The **longjmp** function restores a stack environment and execution locale previously saved in *env* by `setjmp`. `setjmp` and **longjmp** provide a way to execute a nonlocal **goto**; they are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal call and return conventions.

A call to `setjmp` causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point immediately following the corresponding `setjmp` call. Execution resumes as if *value* had just been returned by the `setjmp` call. The values of all variables (except register variables) that are accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable. The value returned by `setjmp` must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.

Microsoft Specific

In Microsoft C++ code on Windows, **longjmp** uses the same stack-unwinding semantics as exception-handling code. It is safe to use in the same places that C++ exceptions can be raised. However, this usage is not portable, and comes with some important caveats.

Only call **longjmp** before the function that called `setjmp` returns; otherwise the results are unpredictable.

Observe the following restrictions when using **longjmp**:

- Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling `setjmp` may not be restored to the proper values after **longjmp** is executed.
- Do not use **longjmp** to transfer control out of an interrupt-handling routine unless the interrupt is caused by a floating-point exception. In this case, a program may return from an interrupt handler via **longjmp** if it first reinitializes the floating-point math package by calling _fpreset.
- Do not use **longjmp** to transfer control from a callback routine invoked directly or indirectly by Windows

code.

- If the code is compiled by using **/EHs** or **/EHsc** and the function that contains the **longjmp** call is **noexcept** then local objects in that function may not be destructed during the stack unwind.

END Microsoft Specific

> **NOTE**
> In portable C++ code, you can't assume `setjmp` and `longjmp` support C++ object semantics. Specifically, a `setjmp` / `longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by **catch** and **throw** would invoke any non-trivial destructors for any automatic objects. In C++ programs, we recommend you use the C++ exception-handling mechanism.

For more information, see Using setjmp and longjmp.

Requirements

ROUTINE	REQUIRED HEADER
longjmp	<setjmp.h>

For additional compatibility information, see Compatibility.

Example

See the example for _fpreset.

See also

Process and Environment Control
setjmp

lrint, lrintf, lrintl, llrint, llrintf, llrintl

11/9/2018 • 2 minutes to read • Edit Online

Rounds the specified floating-point value to the nearest integral value, by using the current rounding mode and direction.

Syntax

```
long int lrint(
   double x
);

long int lrint(
   float x
); //C++ only

long int lrint(
   long double x
); //C++ only

long int lrintf(
   float x
);

long int lrintl(
   long double x
);

long long int llrint(
   double x
);

long long int llrint(
   float x
); //C++ only

long long int llrint(
   long double x
); //C++ only

long long int llrintf(
   float x
);

long long int llrintl(
   long double x
);
```

Parameters

x
the value to round.

Return Value

If successful, returns the rounded integral value of *x*.

ISSUE	RETURN
x is outside the range of the return type *x* = ±∞ *x* = NaN	Raises **FE_INVALID** and returns zero (0).

Remarks

Because C++ allows overloading, you can call overloads of **lrint** and **llrint** that take **float** and **long double** types. In a C program, **lrint** and **llrint** always take a **double**.

If *x* does not represent the floating-point equivalent of an integral value, these functions raise **FE_INEXACT**.

Microsoft specific: When the result is outside the range of the return type, or when the parameter is a NaN or infinity, the return value is implementation defined. The Microsoft compiler returns a zero (0) value.

Requirements

FUNCTION	C HEADER	C++ HEADER
lrint, **lrintf**, **lrintl**, **llrint**, **llrintf**, **llrintl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference

lround, lroundf, lroundl, llround, llroundf, llroundl

10/31/2018 • 2 minutes to read • Edit Online

Rounds a floating-point value to the nearest integer.

Syntax

```
long lround(
   double x
);
long lround(
   float x
);  // C++ only
long lround(
   long double x
);  // C++ only
long lroundf(
   float x
);
long lroundl(
   long double x
);
long long llround(
   double x
);
long long llround(
   float x
);  // C++ only
long long llround(
   long double x
);  // C++ only
long long llroundf(
   float x
);
long long llroundl(
   long double x
);
```

Parameters

x
The floating-point value to round.

Return Value

The **lround** and **llround** functions return the nearest **long** or **long long** integer to *x*. Halfway values are rounded away from zero, regardless of the setting of the floating-point rounding mode. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	none	**_DOMAIN**

Remarks

Because C++ allows overloading, you can call overloads of **lround** or **llround** that take and return **float** and **long double** values. In a C program, **lround** and **llround** always take and return a **double**.

Requirements

ROUTINE	REQUIRED HEADER
lround, **lroundf**, **lroundl**, **llround**, **llroundf**, **llroundl**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_lround.c
// Build with: cl /W4 /Tc crt_lround.c
// This example displays the rounded results of
// the floating-point values 2.499999, -2.499999,
// 2.8, -2.8, 3.5 and -3.5.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.499999;
   float y = 2.8f;
   long double z = 3.5L;

   printf("lround(%f) is %d\n", x, lround(x));
   printf("lround(%f) is %d\n", -x, lround(-x));
   printf("lroundf(%f) is %d\n", y, lroundf(y));
   printf("lroundf(%f) is %d\n", -y, lroundf(-y));
   printf("lroundl(%Lf) is %d\n", z, lroundl(z));
   printf("lroundl(%Lf) is %d\n", -z, lroundl(-z));
}
```

```
lround(2.499999) is 2
lround(-2.499999) is -2
lroundf(2.800000) is 3
lroundf(-2.800000) is -3
lroundl(3.500000) is 4
lroundl(-3.500000) is -4
```

See also

Floating-Point Support
ceil, ceilf, ceill
floor, floorf, floorl
fmod, fmodf
lrint, lrintf, lrintl, llrint, llrintf, llrintl
round, roundf, roundl
nearbyint, nearbyintf, nearbyintl
rint, rintf, rintl

_lrotl, _lrotr

10/31/2018 • 2 minutes to read • Edit Online

Rotates bits to the left (**_lrotl**) or right (**_lrotr**).

Syntax

```
unsigned long _lrotl( unsigned long value, int shift );
unsigned long _lrotr( unsigned long value, int shift );
```

Parameters

value
Value to be rotated.

shift
Number of bits to shift *value*.

Return Value

Both functions return the rotated value. There is no error return.

Remarks

The **_lrotl** and **_lrotr** functions rotate *value* by *shift* bits. **_lrotl** rotates the value left, toward more significant bits. **_lrotr** rotates the value right, toward less significant bits. Both functions wrap bits rotated off one end of *value* to the other end.

Requirements

ROUTINE	REQUIRED HEADER
_lrotl, **_lrotr**	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_lrot.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   unsigned long val = 0x0fac35791;

   printf( "0x%8.8lx rotated left eight bits is 0x%8.8lx\n",
            val, _lrotl( val, 8 ) );
   printf( "0x%8.8lx rotated right four bits is 0x%8.8lx\n",
            val, _lrotr( val, 4 ) );
}
```

```
0xfac35791 rotated left eight bits is 0xc35791fa
0xfac35791 rotated right four bits is 0x1fac3579
```

See also

Floating-Point Support
_rotl, _rotl64, _rotr, _rotr64

lsearch

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _lsearch or security-enhanced _lsearch_s instead.

_lsearch

10/31/2018 • 2 minutes to read • Edit Online

Performs a linear search for a value; adds to end of list if not found. A more secure version of this function is available; see _lsearch_s.

Syntax

```
void *_lsearch(
   const void *key,
   void *base,
   unsigned int *num,
   unsigned int width,
   int (__cdecl *compare)(const void *, const void *)
);
```

Parameters

key
Object to search for.

base
Pointer to the base of array to be searched.

number
Number of elements.

width
Width of each array element.

compare
Pointer to the comparison routine. The first parameter is a pointer to the key for search. The second parameter is a pointer to an array element to be compared with the key.

Return Value

If the key is found, **_lsearch** returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, **_lsearch** returns a pointer to the newly added item at the end of the array.

Remarks

The **_lsearch** function performs a linear search for the value *key* in an array of *number* elements, each of *width* bytes. Unlike **bsearch**, **_lsearch** does not require the array to be sorted. If *key* is not found, **_lsearch** adds it to the end of the array and increments *number*.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **_lsearch** calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. *compare* must compare the elements and return either nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

This function validates its parameters. If *compare*, *key* or *number* is **NULL**, or if *base* is **NULL** and *number* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_lsearch	<search.h>

For more compatibility information, see Compatibility.

Example

```
// crt_lsearch.c
#include <search.h>
#include <string.h>
#include <stdio.h>

int compare( const void *arg1, const void *arg2 );

int main(void)
{
   char * wordlist[4] = { "hello", "thanks", "bye" };
                          // leave room to grow...
   int n = 3;
   char **result;
   char *key = "extra";
   int i;

   printf( "wordlist before _lsearch:" );
   for( i=0; i<n; ++i ) printf( " %s", wordlist[i] );
   printf( "\n" );

   result = (char **)_lsearch( &key, wordlist,
                    &n, sizeof(char *), compare );

   printf( "wordlist after _lsearch:" );
   for( i=0; i<n; ++i ) printf( " %s", wordlist[i] );
   printf( "\n" );
}

int compare(const void *arg1, const void *arg2 )
{
   return( _stricmp( * (char**)arg1, * (char**)arg2 ) );
}
```

```
wordlist before _lsearch: hello thanks bye
wordlist after _lsearch: hello thanks bye extra
```

See also

Searching and Sorting
bsearch
_lfind
_lsearch_s

_lsearch_s

10/31/2018 • 2 minutes to read • Edit Online

Performs a linear search for a value. A version of _lsearch with security enhancements as described in Security Features in the CRT.

Syntax

```
void *_lsearch_s(
   const void *key,
   void *base,
   unsigned int *num,
   size_t size,
   int (__cdecl *compare)(void *, const void *, const void *),
   void * context
);
```

Parameters

key
Object to search for.

base
Pointer to the base of array to be searched.

number
Number of elements.

size
Size of each array element in bytes.

compare
Pointer to the comparison routine. The second parameter is a pointer to the key for search. The third parameter is a pointer to an array element to be compared with the key.

context
A pointer to an object that might be accessed in the comparison function.

Return Value

If *key* is found, **_lsearch_s** returns a pointer to the element of the array at *base* that matches *key*. If *key* is not found, **_lsearch_s** returns a pointer to the newly added item at the end of the array.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, then **errno** is set to **EINVAL** and the function returns **NULL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

KEY	*BASE*	*COMPARE*	*NUMBER*	*SIZE*	ERRNO
NULL	any	any	any	any	**EINVAL**

KEY	*BASE*	*COMPARE*	*NUMBER*	*SIZE*	ERRNO
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	zero	**EINVAL**
any	any	**NULL**	an	any	**EINVAL**

Remarks

The **_lsearch_s** function performs a linear search for the value *key* in an array of *number* elements, each of *width* bytes. Unlike **bsearch_s**, **_lsearch_s** does not require the array to be sorted. If *key* is not found, then **_lsearch_s** adds it to the end of the array and increments *number*.

The *compare* function is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The *compare* function also takes the pointer to the context as the first argument. **_lsearch_s** calls *compare* one or more times during the search, passing pointers to two array elements on each call. *compare* must compare the elements and then return either nonzero (meaning the elements are different) or 0 (meaning the elements are identical).

The *context* pointer can be useful if the searched data structure is part of an object and the *compare* function needs to access members of the object. For example, code in the *compare* function can cast the void pointer into the appropriate object type and access members of that object. The addition of the *context* pointer makes **_lsearch_s** more secure because additional context can be used to avoid reentrancy bugs associated with using static variables to make data available to the *compare* function.

Requirements

ROUTINE	REQUIRED HEADER
_lsearch_s	<search.h>

For more compatibility information, see Compatibility.

See also

Searching and Sorting
bsearch_s
_lfind_s
_lsearch

lseek

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _lseek instead.

_lseek, _lseeki64

11/8/2018 • 2 minutes to read • Edit Online

Moves a file pointer to the specified location.

Syntax

```
long _lseek(
   int fd,
   long offset,
   int origin
);
__int64 _lseeki64(
   int fd,
   __int64 offset,
   int origin
);
```

Parameters

fd
File descriptor referring to an open file.

offset
Number of bytes from *origin*.

origin
Initial position.

Return Value

_lseek returns the offset, in bytes, of the new position from the beginning of the file. **_lseeki64** returns the offset in a 64-bit integer. The function returns -1L to indicate an error. If passed an invalid parameter, such as a bad file descriptor, or the value for *origin* is invalid or the position specified by *offset* is before the beginning of the file, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EBADF** and return -1L. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

For more information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_lseek** function moves the file pointer associated with *fd* to a new location that is *offset* bytes from *origin*. The next operation on the file occurs at the new location. The *origin* argument must be one of the following constants, which are defined in Stdio.h.

ORIGIN VALUE	
SEEK_SET	Beginning of the file.
SEEK_CUR	Current position of the file pointer.

ORIGIN VALUE	
SEEK_END	End of file.

You can use **_lseek** to reposition the pointer anywhere in a file or beyond the end of the file.

Requirements

ROUTINE	REQUIRED HEADER
_lseek	<io.h>
_lseeki64	<io.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_lseek.c
/* This program first opens a file named lseek.txt.
* It then uses _lseek to find the beginning of the file,
* to find the current position in the file, and to find
* the end of the file.
*/

#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <share.h>

int main( void )
{
   int fh;
   long pos;               /* Position of file pointer */
   char buffer[10];

   _sopen_s( &fh, "crt_lseek.c_input", _O_RDONLY, _SH_DENYNO, 0 );

   /* Seek the beginning of the file: */
   pos = _lseek( fh, 0L, SEEK_SET );
   if( pos == -1L )
      perror( "_lseek to beginning failed" );
   else
      printf( "Position for beginning of file seek = %ld\n", pos );

   /* Move file pointer a little */
    _read( fh, buffer, 10 );

   /* Find current position: */
   pos = _lseek( fh, 0L, SEEK_CUR );
   if( pos == -1L )
      perror( "_lseek to current position failed" );
   else
      printf( "Position for current position seek = %ld\n", pos );

   /* Set the end of the file: */
   pos = _lseek( fh, 0L, SEEK_END );
   if( pos == -1L )
      perror( "_lseek to end failed" );
   else
      printf( "Position for end of file seek = %ld\n", pos );

   _close( fh );
}
```

Input: crt_lseek.c_input

```
Line one.
Line two.
Line three.
Line four.
Line five.
```

Output

```
Position for beginning of file seek = 0
Position for current position seek = 10
Position for end of file seek = 57
```

See also

Low-Level I/O
fseek, _fseeki64
_tell, _telli64

_makepath, _wmakepath

10/31/2018 • 2 minutes to read • Edit Online

Create a path name from components. More secure versions of these functions are available; see _makepath_s, _wmakepath_s.

Syntax

```
void _makepath(
   char *path,
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
);
void _wmakepath(
   wchar_t *path,
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
);
```

Parameters

path
Full path buffer.

drive
Contains a letter (A, B, and so on) corresponding to the desired drive and an optional trailing colon. **_makepath** inserts the colon automatically in the composite path if it is missing. If *drive* is **NULL** or points to an empty string, no drive letter appears in the composite *path* string.

dir
Contains the path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either a forward slash (/) or a backslash (\) or both might be used in a single *dir* argument. If no trailing slash (/ or \) is specified, it is inserted automatically. If *dir* is **NULL** or points to an empty string, no directory path is inserted in the composite *path* string.

fname
Contains the base file name without any file name extensions. If *fname* is **NULL** or points to an empty string, no filename is inserted in the composite *path* string.

ext
Contains the actual file name extension, with or without a leading period (.). **_makepath** inserts the period automatically if it does not appear in *ext*. If *ext* is **NULL** or points to an empty string, no extension is inserted in the composite *path* string.

Remarks

The **_makepath** function creates a composite path string from individual components, storing the result in *path*. The *path* might include a drive letter, directory path, filename, and filename extension. **_wmakepath** is a wide-character version of **_makepath**; the arguments to **_wmakepath** are wide-character strings. **_wmakepath** and **_makepath** behave identically otherwise.

Security Note Use a null-terminated string. To avoid buffer overrun, the null-terminated string must not exceed the size of the *path* buffer. **_makepath** does not ensure that the length of the composite path string does not exceed **_MAX_PATH**. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tmakepath	**_makepath**	**_makepath**	**_wmakepath**

The *path* argument must point to an empty buffer large enough to hold the complete path. The composite *path* must be no larger than the **_MAX_PATH** constant, defined in Stdlib.h.

If path is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. In addition, **errno** is set to **EINVAL**. **NULL** values are allowed for all other parameters.

Requirements

ROUTINE	REQUIRED HEADER
_makepath	<stdlib.h>
_wmakepath	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_makepath.c
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath( path_buffer, "c", "\\sample\\crt\\", "makepath", "c" ); // C4996
   // Note: _makepath is deprecated; consider using _makepath_s instead
   printf( "Path created with _makepath: %s\n\n", path_buffer );
   _splitpath( path_buffer, drive, dir, fname, ext ); // C4996
   // Note: _splitpath is deprecated; consider using _splitpath_s instead
   printf( "Path extracted with _splitpath:\n" );
   printf( "   Drive: %s\n", drive );
   printf( "   Dir: %s\n", dir );
   printf( "   Filename: %s\n", fname );
   printf( "   Ext: %s\n", ext );
}
```

```
Path created with _makepath: c:\sample\crt\makepath.c

Path extracted with _splitpath:
   Drive: c:
   Dir: \sample\crt\
   Filename: makepath
   Ext: .c
```

See also

File Handling
_fullpath, _wfullpath
_splitpath, _wsplitpath
_makepath_s, _wmakepath_s

_makepath_s, _wmakepath_s

3/1/2019 • 3 minutes to read • Edit Online

Creates a path name from components. These are versions of _makepath, _wmakepath with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _makepath_s(
   char *path,
   size_t sizeInBytes,
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
);
errno_t _wmakepath_s(
   wchar_t *path,
   size_t sizeInWords,
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
);
template <size_t size>
errno_t _makepath_s(
   char (&path)[size],
   const char *drive,
   const char *dir,
   const char *fname,
   const char *ext
); // C++ only
template <size_t size>
errno_t _wmakepath_s(
   wchar_t (&path)[size],
   const wchar_t *drive,
   const wchar_t *dir,
   const wchar_t *fname,
   const wchar_t *ext
); // C++ only
```

Parameters

path
Full path buffer.

sizeInWords
Size of the buffer in words.

sizeInBytes
Size of the buffer in bytes.

drive
Contains a letter (A, B, and so on) corresponding to the desired drive and an optional trailing colon. **_makepath_s** inserts the colon automatically in the composite path if it is missing. If *drive* is **NULL** or points to an empty string, no drive letter appears in the composite *path* string.

dir

Contains the path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either a forward slash (/) or a backslash (\) or both might be used in a single *dir* argument. If no trailing slash (/ or \) is specified, it is inserted automatically. If *dir* is **NULL** or points to an empty string, no directory path is inserted in the composite *path* string.

fname
Contains the base file name without any file name extensions. If *fname* is **NULL** or points to an empty string, no filename is inserted in the composite *path* string.

ext
Contains the actual file name extension, with or without a leading period (.). **_makepath_s** inserts the period automatically if it does not appear in *ext*. If *ext* is **NULL** or points to an empty string, no extension is inserted in the composite *path* string.

Return Value

Zero if successful; an error code on failure.

Error Conditions

PATH	*SIZEINWORDS / SIZEINBYTES*	RETURN	CONTENTS OF *PATH*
NULL	any	**EINVAL**	not modified
any	<= 0	**EINVAL**	not modified

If any of the above error conditions occurs, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the functions returns **EINVAL**. **NULL** is allowed for the parameters *drive*, *fname*, and *ext*. For information about the behavior when these parameters are null pointers or empty strings, see the Remarks section.

Remarks

The **_makepath_s** function creates a composite path string from individual components, storing the result in *path*. The *path* might include a drive letter, directory path, file name, and file name extension. **_wmakepath_s** is a wide-character version of **_makepath_s**; the arguments to **_wmakepath_s** are wide-character strings. **_wmakepath_s** and **_makepath_s** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tmakepath_s	**_makepath_s**	**_makepath_s**	**_wmakepath_s**

The *path* argument must point to an empty buffer large enough to hold the complete path. The composite *path* must be no larger than the **_MAX_PATH** constant, defined in Stdlib.h.

If path is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. In addition, **errno** is set to **EINVAL**. **NULL** values are allowed for all other parameters.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

ROUTINE	REQUIRED HEADER
_makepath_s	<stdlib.h>
_wmakepath_s	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_makepath_s.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];
   errno_t err;

   err = _makepath_s( path_buffer, _MAX_PATH, "c", "\\sample\\crt\\",
                      "crt_makepath_s", "c" );
   if (err != 0)
   {
      printf("Error creating path. Error code %d.\n", err);
      exit(1);
   }
   printf( "Path created with _makepath_s: %s\n\n", path_buffer );
   err = _splitpath_s( path_buffer, drive, _MAX_DRIVE, dir, _MAX_DIR, fname,
                       _MAX_FNAME, ext, _MAX_EXT );
   if (err != 0)
   {
      printf("Error splitting the path. Error code %d.\n", err);
      exit(1);
   }
   printf( "Path extracted with _splitpath_s:\n" );
   printf( "   Drive: %s\n", drive );
   printf( "   Dir: %s\n", dir );
   printf( "   Filename: %s\n", fname );
   printf( "   Ext: %s\n", ext );
}
```

```
Path created with _makepath_s: c:\sample\crt\crt_makepath_s.c

Path extracted with _splitpath_s:
   Drive: c:
   Dir: \sample\crt\
   Filename: crt_makepath_s
   Ext: .c
```

See also

File Handling
_fullpath, _wfullpath

_splitpath_s, _wsplitpath_s
_makepath, _wmakepath

malloc

10/31/2018 • 3 minutes to read • Edit Online

Allocates memory blocks.

Syntax

```
void *malloc(
   size_t size
);
```

Parameters

size
Bytes to allocate.

Return Value

malloc returns a void pointer to the allocated space, or **NULL** if there is insufficient memory available. To return a pointer to a type other than **void**, use a type cast on the return value. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object that has an alignment requirement less than or equal to that of the fundamental alignment. (In Visual C++, the fundamental alignment is the alignment that's required for a **double**, or 8 bytes. In code that targets 64-bit platforms, it's 16 bytes.) Use _aligned_malloc to allocate storage for objects that have a larger alignment requirement—for example, the SSE types __m128 and __**m256**, and types that are declared by using `__declspec(align( n ))` where **n** is greater than 8. If *size* is 0, **malloc** allocates a zero-length item in the heap and returns a valid pointer to that item. Always check the return from **malloc**, even if the amount of memory requested is small.

Remarks

The **malloc** function allocates a memory block of at least *size* bytes. The block may be larger than *size* bytes because of the space that's required for alignment and maintenance information.

malloc sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The startup code uses **malloc** to allocate storage for the **_environ**, *envp*, and *argv* variables. The following functions and their wide-character counterparts also call **malloc**.

calloc	fscanf	_getw	setvbuf
_exec functions	fseek	_popen	_spawn functions
fgetc	fsetpos	printf	_strdup
_fgetchar	_fullpath	putc	system

fgets	fwrite	putchar	_tempnam
fprintf	getc	_putenv	ungetc
fputc	getchar	puts	vfprintf
_fputchar	_getcwd	_putw	vprintf
fputs	_getdcwd	scanf	
fread	gets	_searchenv	

The C++ _set_new_mode function sets the new handler mode for **malloc**. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call `_set_new_mode(1)` early in your program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **malloc** resolves to _malloc_dbg. For more information about how the heap is managed during the debugging process, see CRT Debug Heap Details.

malloc is marked `__declspec(noalias)` and `__declspec(restrict)`; this means that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

ROUTINE	REQUIRED HEADER
malloc	<stdlib.h> and <malloc.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_malloc.c
// This program allocates memory with
// malloc, then frees the memory with free.

#include <stdlib.h>   // For _MAX_PATH definition
#include <stdio.h>
#include <malloc.h>

int main( void )
{
   char *string;

   // Allocate space for a path name
   string = malloc( _MAX_PATH );

   // In a C++ file, explicitly cast malloc's return.  For example,
   // string = (char *)malloc( _MAX_PATH );

   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
   {
      printf( "Memory space allocated for path name\n" );
      free( string );
      printf( "Memory freed\n" );
   }
}
```

```
Memory space allocated for path name
Memory freed
```

See also

Memory Allocation
calloc
free
realloc
_aligned_malloc

_malloc_dbg

10/31/2018 • 2 minutes to read • Edit Online

Allocates a block of memory in the heap with additional space for a debugging header and overwrite buffers (debug version only).

Syntax

```
void *_malloc_dbg(
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

size
Requested size of the memory block (in bytes).

blockType
Requested type of the memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested the allocation operation or **NULL**.

linenumber
Line number in the source file where the allocation operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_malloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function returns a pointer to the user portion of the allocated memory block, calls the new handler function, or returns **NULL**. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the malloc function.

Remarks

_malloc_dbg is a debug version of the malloc function. When _DEBUG is not defined, each call to **_malloc_dbg** is reduced to a call to **malloc**. Both **malloc** and **_malloc_dbg** allocate a block of memory in the base heap, but **_malloc_dbg** offers several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename*/*linenumber* information to determine the origin of allocation requests.

_malloc_dbg allocates the memory block with slightly more space than the requested *size*. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. When the block is allocated, the user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

_malloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory needed

(including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_malloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

For a sample of how to use **_malloc_dbg**, see crt_dbg1.

See also

Debug Routines
malloc
_calloc_dbg
_calloc_dbg

_malloca

2/7/2019 • 3 minutes to read • Edit Online

Allocates memory on the stack. This is a version of _alloca with security enhancements as described in Security Features in the CRT.

Syntax

```
void *_malloca(
   size_t size
);
```

Parameters

size
Bytes to be allocated from the stack.

Return Value

The **_malloca** routine returns a **void** pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. If *size* is 0, **_malloca** allocates a zero-length item and returns a valid pointer to that item.

If *size* is greater than **_ALLOCA_S_THRESHOLD**, then **_malloca** attempts to allocate on the heap, and returns a null pointer if the space can't be allocated. If *size* is less than or equal to **_ALLOCA_S_THRESHOLD**, then **_malloca** attempts to allocate on the stack, and a stack overflow exception is generated if the space can't be allocated. The stack overflow exception isn't a C++ exception; it's a structured exception. Instead of using C++ exception handling, you must use Structured Exception Handling (SEH) to catch this exception.

Remarks

_malloca allocates *size* bytes from the program stack or the heap if the request exceeds a certain size in bytes given by **_ALLOCA_S_THRESHOLD**. The difference between **_malloca** and **_alloca** is that **_alloca** always allocates on the stack, regardless of the size. Unlike **_alloca**, which does not require or permit a call to **free** to free the memory so allocated, **_malloca** requires the use of _freea to free memory. In debug mode, **_malloca** always allocates memory from the heap.

There are restrictions to explicitly calling **_malloca** in an exception handler (EH). EH routines that run on x86-class processors operate in their own memory frame: They perform their tasks in memory space that is not based on the current location of the stack pointer of the enclosing function. The most common implementations include Windows NT structured exception handling (SEH) and C++ catch clause expressions. Therefore, explicitly calling **_malloca** in any of the following scenarios results in program failure during the return to the calling EH routine:

- Windows NT SEH exception filter expression: **__except** (`_malloca ()`)
- Windows NT SEH final exception handler: **__finally** { `_malloca ()` }
- C++ EH catch clause expression

However, **_malloca** can be called directly from within an EH routine or from an application-supplied callback that gets invoked by one of the EH scenarios previously listed.

IMPORTANT

In Windows XP, if **_malloca** is called inside a try/catch block, you must call _resetstkoflw in the catch block.

In addition to the above restrictions, when using the /clr (Common Language Runtime Compilation) option, **_malloca** cannot be used in **__except** blocks. For more information, see /clr Restrictions.

Requirements

ROUTINE	REQUIRED HEADER
_malloca	<malloc.h>

Example

```
// crt_malloca_simple.c
#include <stdio.h>
#include <malloc.h>

void Fn()
{
   char * buf = (char *)_malloca( 100 );
   // do something with buf
   _freea( buf );
}

int main()
{
   Fn();
}
```

Example

```
// crt_malloca_exception.c
// This program demonstrates the use of
// _malloca and trapping any exceptions
// that may occur.

#include <windows.h>
#include <stdio.h>
#include <malloc.h>

int main()
{
    int     size;
    int     numberRead = 0;
    int     errcode = 0;
    void    *p = NULL;
    void    *pMarker = NULL;

    while (numberRead == 0)
    {
        printf_s("Enter the number of bytes to allocate "
                 "using _malloca: ");
        numberRead = scanf_s("%d", &size);
    }

    // Do not use try/catch for _malloca,
    // use __try/__except, since _malloca throws
    // Structured Exceptions, not C++ exceptions.

    __try
    {
        if (size > 0)
        {
            p =  _malloca( size );
        }
        else
        {
            printf_s("Size must be a positive number.");
        }
        _freea( p );
    }

    // Catch any exceptions that may occur.
    __except( GetExceptionCode() == STATUS_STACK_OVERFLOW )
    {
        printf_s("_malloca failed!\n");

        // If the stack overflows, use this function to restore.
        errcode = _resetstkoflw();
        if (errcode)
        {
            printf("Could not reset the stack!");
            _exit(1);
        }
    };
}
```

Input

```
1000
```

Sample Output

```
Enter the number of bytes to allocate using _malloca: 1000
```

See also

Memory Allocation
calloc
malloc
realloc
_resetstkoflw

_matherr

11/8/2018 • 3 minutes to read • Edit Online

Handles math errors.

Syntax

```
int _matherr( struct _exception * except );
```

Parameters

except
Pointer to the structure containing error information.

Return Value

_matherr returns 0 to indicate an error, or a nonzero value to indicate success. If **_matherr** returns 0, an error message can be displayed and **errno** is set to an appropriate error value. If **_matherr** returns a nonzero value, no error message is displayed and **errno** remains unchanged.

For more information about return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_matherr** function processes errors generated by the floating-point functions of the math library. These functions call **_matherr** when an error is detected.

For special error handling, you can provide a different definition of **_matherr**. If you use the dynamically linked version of the C run-time library (CRT), you can replace the default **_matherr** routine in a client executable with a user-defined version. However, you cannot replace the default **_matherr** routine in a DLL client of the CRT DLL.

When an error occurs in a math routine, **_matherr** is called with a pointer to an **_exception** type structure (defined in <math.h>) as an argument. The **_exception** structure contains the following elements.

```
struct _exception
{
    int    type;   // exception type - see below
    char*  name;   // name of function where error occurred
    double arg1;   // first argument to function
    double arg2;   // second argument (if any) to function
    double retval; // value to be returned by function
};
```

The **type** member specifies the type of math error. It is one of the following values, defined in <math.h>:

MACRO	MEANING
_DOMAIN	Argument domain error
_SING	Argument singularity

MACRO	MEANING
_OVERFLOW	Overflow range error
_PLOSS	Partial loss of significance
_TLOSS	Total loss of significance
_UNDERFLOW	The result is too small to be represented. (This condition is not currently supported.)

The structure member **name** is a pointer to a null-terminated string containing the name of the function that caused the error. The structure members **arg1** and **arg2** specify the values that caused the error. If only one argument is given, it is stored in **arg1**.

The default return value for the given error is **retval**. If you change the return value, it must specify whether an error actually occurred.

Requirements

ROUTINE	REQUIRED HEADER
_matherr	<math.h>

For more compatibility information, see Compatibility.

Example

```
// crt_matherr.c
/* illustrates writing an error routine for math
* functions. The error function must be:
*      _matherr
*/

#include <math.h>
#include <string.h>
#include <stdio.h>

int main()
{
    /* Do several math operations that cause errors. The _matherr
     * routine handles _DOMAIN errors, but lets the system handle
     * other errors normally.
     */
    printf( "log( -2.0 ) = %e\n", log( -2.0 ) );
    printf( "log10( -5.0 ) = %e\n", log10( -5.0 ) );
    printf( "log( 0.0 ) = %e\n", log( 0.0 ) );
}

/* Handle several math errors caused by passing a negative argument
* to log or log10 (_DOMAIN errors). When this happens, _matherr
* returns the natural or base-10 logarithm of the absolute value
* of the argument and suppresses the usual error message.
*/
int _matherr( struct _exception *except )
{
    /* Handle _DOMAIN errors for log or log10. */
    if( except->type == _DOMAIN )
    {
        if( strcmp( except->name, "log" ) == 0 )
        {
            except->retval = log( -(except->arg1) );
            printf( "Special: using absolute value: %s: _DOMAIN "
                    "error\n", except->name );
            return 1;
        }
        else if( strcmp( except->name, "log10" ) == 0 )
        {
            except->retval = log10( -(except->arg1) );
            printf( "Special: using absolute value: %s: _DOMAIN "
                    "error\n", except->name );
            return 1;
        }
    }
    printf( "Normal: " );
    return 0;    /* Else use the default actions */
}
```

```
Special: using absolute value: log: _DOMAIN error
log( -2.0 ) = 6.931472e-01
Special: using absolute value: log10: _DOMAIN error
log10( -5.0 ) = 6.989700e-01
Normal: log( 0.0 ) = -inf
```

See also

Floating-Point Support

__max

10/31/2018 • 2 minutes to read • Edit Online

A preprocessor macro that returns the larger of two values.

Syntax

```
#define __max(a,b) (((a) > (b)) ? (a) : (b))
```

Parameters

a, *b*
Values of any numeric type to be compared.

Return Value

__max returns the larger of its arguments.

Remarks

The **__max** macro compares two values and returns the value of the larger one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

The argument returned is evaluated twice by the macro. This can lead to unexpected results if the argument is an expression that alters its value when it is evaluated, such as `*p++`.

Requirements

MACRO	REQUIRED HEADER
__max	<stdlib.h>

Example

For more information, see the example for __min.

See also

Floating-Point Support
__min

_mbbtombc, _mbbtombc_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a single-byte multibyte character to a corresponding double-byte multibyte character.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _mbbtombc(
   unsigned int c
);
unsigned int _mbbtombc_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Single-byte character to convert.

locale
Locale to use.

Return Value

If **_mbbtombc** successfully converts *c*, it returns a multibyte character; otherwise, it returns *c*.

Remarks

The **_mbbtombc** function converts a given single-byte multibyte character to a corresponding double-byte multibyte character. Characters must be within the range 0x20 - 0x7E or 0xA1 - 0xDF to be converted.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of this function are identical, except that **_mbbtombc** uses the current locale for this locale-dependent behavior and **_mbbtombc_l** instead uses the locale parameter that's passed in. For more information, see Locale.

In earlier versions, **_mbbtombc** was named **hantozen**. For new code, use **_mbbtombc**.

Requirements

ROUTINE	REQUIRED HEADER
_mbbtombc	<mbstring.h>
_mbbtombc_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Data Conversion
_mbctombb, _mbctombb_l

_mbbtype, _mbbtype_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the byte type, based on the previous byte.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _mbbtype(
   unsigned char c,
   int type
);
int _mbbtype_l(
   unsigned char c,
   int type,
   _locale_t locale
);
```

Parameters

c
The character to test.

type
The type of byte to test for.

locale
The locale to use.

Return Value

_mbbtype returns the type of byte in a string. This decision is context-sensitive, as specified by the value of *type*, which provides the control test condition. *type* is the type of the previous byte in the string. The manifest constants in the following table are defined in Mbctype.h.

VALUE OF *TYPE*	_MBBTYPE TESTS FOR	RETURN VALUE	*C*
Any value except 1	Valid single byte or lead byte	**_MBC_SINGLE** (0)	Single byte (0x20 - 0x7E, 0xA1 - 0xDF)
Any value except 1	Valid single byte or lead byte	**_MBC_LEAD** (1)	Lead byte of multibyte character (0x81 - 0x9F, 0xE0 - 0xFC)
Any value except 1	Valid single-byte or lead byte	**_MBC_ILLEGAL** (-1)	Invalid character (any value except 0x20 - 0x7E, 0xA1 - 0xDF, 0x81 - 0x9F, 0xE0 - 0xFC

VALUE OF *TYPE*	_MBBTYPE TESTS FOR	RETURN VALUE	*C*
1	Valid trail byte	**_MBC_TRAIL** (2)	Trailing byte of multibyte character (0x40 - 0x7E, 0x80 - 0xFC)
1	Valid trail byte	**_MBC_ILLEGAL** (-1)	Invalid character (any value except 0x20 - 0x7E, 0xA1 - 0xDF, 0x81 - 0x9F, 0xE0 - 0xFC

Remarks

The **_mbbtype** function determines the type of a byte in a multibyte character. If the value of *type* is any value except 1, **_mbbtype** tests for a valid single-byte or lead byte of a multibyte character. If the value of *type* is 1, **_mbbtype** tests for a valid trail byte of a multibyte character.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The **_mbbtype** version of this function uses the current locale for this locale-dependent behavior; the **_mbbtype_l** version is identical except that it use the locale parameter that's passed in instead. For more information, see Locale.

In earlier versions, **_mbbtype** was named **chkctype**. For new code, use **_mbbtype** instead.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_mbbtype	<mbstring.h>	<mbctype.h>*
_mbbtype_l	<mbstring.h>	<mbctype.h>*

* For definitions of manifest constants that are used as return values.

For more compatibility information, see Compatibility.

See also

Byte Classification

_mbccpy, _mbccpy_l

10/31/2018 • 2 minutes to read • Edit Online

Copies a multibyte character from one string to another string. More secure versions of these functions are available; see _mbccpy_s, _mbccpy_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
void _mbccpy(
   unsigned char *dest,
   const unsigned char *src
);
void _mbccpy_l(
   unsigned char *dest,
   const unsigned char *src,
   _locale_t locale
);
```

Parameters

dest
Copy destination.

src
Multibyte character to copy.

locale
Locale to use.

Remarks

The **_mbccpy** function copies one multibyte character from *src* to *dest*.

This function validates its parameters. If **_mbccpy** is passed a null pointer for *dest* or *src*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL**.

_mbccpy uses the current locale for any locale-dependent behavior. **_mbccpy_l** is identical to **_mbccpy** except that **_mbccpy_l** uses the locale passed in for any locale-dependent behavior. For more information, see Locale.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tccpy	Maps to macro or inline function	**_mbccpy**	Maps to macro or inline function
_tccpy_l	n/a	**_mbccpy_l**	n/a

Requirements

ROUTINE	REQUIRED HEADER
_mbccpy	<mbctype.h>
_mbccpy_l	<mbctype.h>

For more compatibility information, see Compatibility.

See also

Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l

_mbccpy_s, _mbccpy_s_l

10/31/2018 • 2 minutes to read • Edit Online

Copies one multibyte character from a string to another string. These versions of _mbccpy, _mbccpy_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _mbccpy_s(
   unsigned char *dest,
   size_t buffSizeInBytes,
   int * pCopied,
   const unsigned char *src
);
errno_t _mbccpy_s_l(
   unsigned char *dest,
   size_t buffSizeInBytes,
   int * pCopied,
   const unsigned char *src,
   locale_t locale
);
template <size_t size>
errno_t _mbccpy_s(
   unsigned char (&dest)[size],
   int * pCopied,
   const unsigned char *src
); // C++ only
template <size_t size>
errno_t _mbccpy_s_l(
   unsigned char (&dest)[size],
   int * pCopied,
   const unsigned char *src,
   locale_t locale
); // C++ only
```

Parameters

dest
Copy destination.

buffSizeInBytes
Size of the destination buffer.

pCopied
Filled with the number of bytes copied (1 or 2 if successful). Pass **NULL** if you don't care about the number.

src
Multibyte character to copy.

locale
Locale to use.

Return Value

Zero if successful; an error code on failure. If *src* or *dest* is **NULL**, or if more than **buffSizeinBytes** bytes would be copied to *dest*, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return **EINVAL** and **errno** is set to **EINVAL**.

Remarks

The **_mbccpy_s** function copies one multibyte character from *src* to *dest*. If *src* does not point to the lead byte of a multibyte character as determined by an implicit call to _ismbblead, then the single byte that *src* points to is copied. If *src* points to a lead byte but the following byte is 0 and thus invalid, then 0 is copied to *dest*, **errno** is set to **EILSEQ**, and the function returns **EILSEQ**.

_mbccpy_s does not append a null terminator; however, if *src* points to a null character, then that null is copied to *dest* (this is just a regular single-byte copy).

The value in *pCopied* is filled with the number of bytes copied. Possible values are 1 and 2 if the operation is successful. If **NULL** is passed in, this parameter is ignored.

SRC	COPIED TO *DEST*	*PCOPIED*	RETURN VALUE
non-lead-byte	non-lead-byte	1	0
0	0	1	0
lead-byte followed by non-0	lead-byte followed by non-0	2	0
lead-byte followed by 0	0	1	**EILSEQ**

Note that the second row is just a special case of the first. Also note that the table assumes *buffSizeInBytes* >= *pCopied*.

_mbccpy_s uses the current locale for any locale-dependent behavior. **_mbccpy_s_l** is identical to **_mbccpy_s** except that **_mbccpy_s_l** uses the locale passed in for any locale-dependent behavior.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tccpy_s	Maps to macro or inline function.	**_mbccpy_s**	Maps to macro or inline function.

Requirements

ROUTINE	REQUIRED HEADER
_mbccpy_s	<mbstring.h>
_mbccpy_s_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l

_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l

10/31/2018 • 2 minutes to read • Edit Online

Converts between Japan Industry Standard (JIS) and Japan Microsoft (JMS) characters.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _mbcjistojms(
   unsigned int c
);
unsigned int _mbcjistojms_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbcjmstojis(
   unsigned int c
);
unsigned int _mbcjmstojis_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Character to convert.

locale
Locale to use.

Return Value

On Japanese locale, these functions return a converted character or return 0 if no conversion is possible. On a non-Japanese locale, these functions return the character passed in.

Remarks

The **_mbcjistojms** function converts a Japan Industry Standard (JIS) character to a Microsoft Kanji (Shift JIS) character. The character is converted only if the lead and trail bytes are in the range 0x21 - 0x7E. If the lead or trial byte is outside this range, **errno** is set to **EILSEQ**. For more information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

The **_mbcjmstojis** function converts a Shift JIS character to a JIS character. The character is converted only if the lead byte is in the range 0x81 - 0x9F or 0xE0 - 0xFC and the trail byte is in the range 0x40 - 0x7E or 0x80 - 0xFC. Note that some code points in that range do not have a character assigned and so cannot be converted.

The value *c* should be a 16-bit value whose upper 8 bits represent the lead byte of the character to convert and whose lower 8 bits represent the trail byte.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In earlier versions, **_mbcjistojms** and **_mbcjmstojis** were called **jistojms** and **jmstojis**, respectively. **_mbcjistojms**, **_mbcjistojms_l**, **_mbcjmstojis** and **_mbcjmstojis_l** should be used instead.

Requirements

ROUTINE	REQUIRED HEADER
_mbcjistojms	<mbstring.h>
_mbcjistojms_l	<mbstring.h>
_mbcjmstojis	<mbstring.h>
_mbcjmstojis_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Data Conversion
_ismbb Routines

_mbclen, mblen, _mblen_l, _mbclen_l

2/4/2019 • 2 minutes to read • Edit Online

Gets the length and determines the validity of a multibyte character.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t _mbclen(
   const unsigned char *c
);
size_t _mbclen_l(
   unsigned char const* c,
   _locale_t locale
);
int mblen(
   const char *mbstr,
   size_t count
);
int _mblen_l(
   const char *mbstr,
   size_t count,
   _locale_t locale
);
```

Parameters

c
Multibyte character.

mbstr
Address of a multibyte-character byte sequence.

count
Number of bytes to check.

locale
Locale to use.

Return Value

_mbclen returns 1 or 2, according to whether the multibyte character *c* is 1 or 2 bytes long. There is no error return for **_mbclen**. If *mbstr* isn't **NULL**, **mblen** returns the length, in bytes, of the multibyte character. If *mbstr* is **NULL** or it points to the wide-character null character, **mblen** returns 0. When the object that *mbstr* points to doesn't form a valid multibyte character within the first *count* characters, **mblen** returns -1.

Remarks

The **_mbclen** function returns the length, in bytes, of the multibyte character *c*. If *c* doesn't point to the lead

byte of a multibyte character as determined by an implicit call to **_ismbblead**, the result of **_mbclen** is unpredictable.

mblen returns the length in bytes of *mbstr* if it's a valid multibyte character and determines multibyte-character validity associated with the code page. **mblen** examines *count* or fewer bytes contained in *mbstr*, but not more than **MB_CUR_MAX** bytes.

The output value is affected by the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior. The **_l** suffixed versions behave the same, but they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tclen	Maps to macro or inline function	**_mbclen**	Maps to macro or inline function

Requirements

ROUTINE	REQUIRED HEADER
_mbclen	<mbstring.h>
mblen	<stdlib.h>
_mblen_l	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mblen.c
/* illustrates the behavior of the mblen function
*/

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int      i;
    char    *pmbc = (char *)malloc( sizeof( char ) );
    wchar_t  wc   = L'a';

    printf( "Convert wide character to multibyte character:\n" );
    wctomb_s( &i, pmbc, sizeof(char), wc );
    printf( "   Characters converted: %u\n", i );
    printf( "   Multibyte character: %x\n\n", *pmbc );

    i = mblen( pmbc, MB_CUR_MAX );
    printf( "Length in bytes of multibyte character %x: %u\n", *pmbc, i );

    pmbc = NULL;
    i = mblen( pmbc, MB_CUR_MAX );
    printf( "Length in bytes of NULL multibyte character %x: %u\n", pmbc, i );
}
```

```
Convert wide character to multibyte character:
   Characters converted: 1
   Multibyte character: 61

Length in bytes of multibyte character 61: 1
Length in bytes of NULL multibyte character 0: 0
```

See also

Character Classification
Locale
Interpretation of Multibyte-Character Sequences
_mbccpy, _mbccpy_l
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l

10/31/2018 • 2 minutes to read • Edit Online

Converts between hiragana and katakana characters.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _mbctohira(
   unsigned int c
);
unsigned int _mbctohira_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbctokata(
   unsigned int c
);
unsigned int _mbctokata_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Multibyte character to convert.

locale
Locale to use.

Return Value

Each of these functions returns the converted character *c*, if possible. Otherwise it returns the character *c* unchanged.

Remarks

The **_mbctohira** and **_mbctokata** functions test a character *c* and, if possible, apply one of the following conversions.

ROUTINES	CONVERTS
_mbctohira, **_mbctohira_l**	Multibyte katakana to multibyte hiragana.
_mbctokata, **_mbctokata_l**	Multibyte hiragana to multibyte katakana.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more

information. The versions of these functions are identical, except that the ones that don't have the **_l** suffix use the current locale for this locale-dependent behavior and the ones that do have the **_l** suffix instead use the locale parameter that's passed in. For more information, see Locale.

In earlier versions, **_mbctohira** was named **jtohira** and **_mbctokata** was named **jtokata**. For new code, use the new names.

Requirements

ROUTINE	REQUIRED HEADER
_mbctohira	<mbstring.h>
_mbctohira_l	<mbstring.h>
_mbctokata	<mbstring.h>
_mbctokata_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Data Conversion
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l
_mbctombb, _mbctombb_l

_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

10/31/2018 • 2 minutes to read • Edit Online

Tests and converts the case of a multibyte character.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _mbctolower(
   unsigned int c
);
unsigned int _mbctolower_l(
   unsigned int c,
   _locale_t locale
);
unsigned int _mbctoupper(
   unsigned int c
);
unsigned int _mbctoupper_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Multibyte character to convert.

locale
Locale to use.

Return Value

Each of these functions returns the converted character *c*, if possible. Otherwise it returns the character *c* unchanged.

Remarks

The functions test a character *c* and, if possible, apply one of the following conversions.

ROUTINES	CONVERTS
_mbctolower, **_mbctolower_l**	Uppercase character to lowercase character.
_mbctoupper, **_mbctoupper_l**	Lowercase character to uppercase character.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it uses the locale parameter passed in instead. For more information, see Locale.

In previous versions, **_mbctolower** was called **jtolower**, and **_mbctoupper** was called **jtoupper**. For new code, use the new names instead.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_totlower	**tolower**	**_mbctolower**	**towlower**
_totlower_l	**_tolower_l**	**_mbctolower_l**	**_towlower_t**
_totupper	**toupper**	**_mbctoupper**	**towupper**
_totupper_l	**toupper_l**	**_mbctoupper_l**	**_towupper_l**

Requirements

ROUTINES	REQUIRED HEADER
_mbctolower, **_mbctolower_l**	<mbstring.h>
_mbctoupper, **_mbctoupper_l**	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Data Conversion
_mbbtombc, _mbbtombc_l
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l
_mbctombb, _mbctombb_l

_mbctombb, _mbctombb_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a double-byte multibyte character to a corresponding single-byte multibyte character.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _mbctombb(
   unsigned int c
);
unsigned int _mbctombb_l(
   unsigned int c,
   _locale_t locale
);
```

Parameters

c
Multibyte character to convert.

locale
Locale to use.

Return Value

If successful, **_mbctombb** and **_mbctombb_l** returns the single-byte character that corresponds to *c*; otherwise it returns *c*.

Remarks

The **_mbctombb** and **_mbctombb_l** functions convert a given multibyte character to a corresponding single-byte multibyte character. Characters must correspond to single-byte characters within the range 0x20 - 0x7E or 0xA1 - 0xDF to be converted.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it use the locale parameter passed in instead. For more information, see Locale.

In previous versions, **_mbctombb** was called **zentohan**. Use **_mbctombb** instead.

Requirements

ROUTINE	REQUIRED HEADER
_mbctombb	<mbstring.h>

ROUTINE	REQUIRED HEADER
_mbctombb_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

Data Conversion
_mbbtombc, _mbbtombc_l
_mbcjistojms, _mbcjistojms_l, _mbcjmstojis, _mbcjmstojis_l
_mbctohira, _mbctohira_l, _mbctokata, _mbctokata_l
_mbctolower, _mbctolower_l, _mbctoupper, _mbctoupper_l

mbrlen

11/9/2018 • 2 minutes to read • Edit Online

Determine the number of bytes that are required to complete a multibyte character in the current locale, with the capability of restarting in the middle of a multibyte character.

Syntax

```
size_t mbrlen(
   const char * str,
   size_t count,
   mbstate_t * mbstate
);
```

Parameters

str
Pointer to the next byte to inspect in a multibyte character string.

count
The maximum number of bytes to inspect.

mbstate
Pointer to the current shift state of the initial byte of *str*.

Return Value

One of the following values:

0	The next *count* or fewer bytes complete the multibyte character that represents the wide null character.
1 to *count*, inclusive	The next *count* or fewer bytes complete a valid multibyte character. The value returned is the number of bytes that complete the multibyte character.
(size_t)(-2)	The next *count* bytes contribute to an incomplete but potentially valid multibyte character and all *count* bytes have been processed.
(size_t)(-1)	An encoding error occurred. The next *count* or fewer bytes do not contribute to a complete and valid multibyte character. In this case, **errno** is set to EILSEQ and the conversion state in *mbstate* is unspecified.

Remarks

The **mbrlen** function inspects at most *count* bytes starting with the byte pointed to by *str* to determine the number of bytes that are required to complete the next multibyte character, including any shift sequences. It is equivalent to the call `mbrtowc(NULL, str, count, &mbstate)` where *mbstate* is either a user-provided **mbstate_t** object, or a static internal object provided by the library.

The **mbrlen** function saves and uses the shift state of an incomplete multibyte character in the *mbstate* parameter. This gives **mbrlen** the capability of restarting in the middle of a multibyte character if need be, examining at most *count* bytes. If *mbstate* is a null pointer, **mbrlen** uses an internal, static **mbstate_t** object to store the shift state. Because the internal **mbstate_t** object is not thread-safe, we recommend that you always allocate and pass your own *mbstate* parameter.

The **mbrlen** function differs from _mbclen, mblen, _mblen_l by its restartability. The shift state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application should use **wcsrlen** instead of **wcslen** if a subsequent call to **wcsrtombs** is used instead of **wcstombs**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
not applicable	not applicable	**mbrlen**	not applicable

Requirements

ROUTINE	REQUIRED HEADER
mbrlen	<wchar.h>

For additional compatibility information, see Compatibility.

Example

This example shows how the interpretation of multibyte characters depends on the current code page, and demonstrates the resuming capability of **mbrlen**.

```
// crt_mbrlen.c
// Compile by using: cl crt_mbrlen.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <wchar.h>

size_t Example(const char * pStr)
{
    size_t      charLen = 0;
    size_t      charCount = 0;
    mbstate_t   mbState = {0};

    while ((charLen = mbrlen(pStr++, 1, &mbState)) != 0 &&
            charLen != (size_t)-1)
    {
        if (charLen != (size_t)-2) // if complete mbcs char,
        {
            charCount++;
        }
    }
    return (charCount);
}

int main( void )
{
    int         cp;
    size_t      charCount = 0;
    const char  *pSample =
        "\x82\xD0\x82\xE7\x82\xAA\x82\xC8: Shift-jis hiragana.";

    cp = _getmbcp();
    charCount = Example(pSample);
    printf("\nCode page: %d\n%s\nCharacter count: %d\n",
        cp, pSample, charCount);

    setlocale(LC_ALL, "ja-JP"); // Set Japanese locale
    _setmbcp(932); // and Japanese multibyte code page
    cp = _getmbcp();
    charCount = Example(pSample);
    printf("\nCode page: %d\n%s\nCharacter count: %d\n",
        cp, pSample, charCount);
}
```

```

Code page: 0
é╨éτé¬é╚: Shift-jis hiragana.
Character count: 29

Code page: 932
????: Shift-jis hiragana.
Character count: 25
```

See also

String Manipulation
Locale

mbrtoc16, mbrtoc32

11/9/2018 • 3 minutes to read • Edit Online

Translates the first multibyte character in a narrow string into the equivalent UTF-16 or UTF-32 character.

Syntax

```
size_t mbrtoc16(
   char16_t* destination,
   const char* source,
   size_t max_bytes,
   mbstate_t* state
);

size_t mbrtoc32(
   char32_t* destination,
   const char* source,
   size_t max_bytes,
   mbstate_t* state
);
```

Parameters

destination
Pointer to the **char16_t** or **char32_t** equivalent of the multibyte character to convert. If null, the function does not store a value.

source
Pointer to the multibyte character string to convert.

max_bytes
The maximum number of bytes in *source* to examine for a character to convert. This should be a value between one and the number of bytes, including any null terminator, remaining in *source*.

state
Pointer to a **mbstate_t** conversion state object used to interpret the multibyte string to one or more output characters.

Return Value

On success, returns the value of the first of these conditions that applies, given the current *state* value:

VALUE	CONDITION
0	The next *max_bytes* or fewer characters converted from *source* correspond to the null wide character, which is the value stored if *destination* is not null. *state* contains the initial shift state.
Between 1 and *max_bytes*, inclusive	The value returned is the number of bytes of *source* that complete a valid multibyte character. The converted wide character is stored if *destination* is not null.

VALUE	CONDITION
-3	The next wide character resulting from a previous call to the function has been stored in *destination* if *destination* is not null. No bytes from *source* are consumed by this call to the function. When *source* points to a multibyte character that requires more than one wide character to represent (for example, a surrogate pair), then the *state* value is updated so that the next function call writes out the additional character.
-2	The next *max_bytes* bytes represent an incomplete, but potentially valid, multibyte character. No value is stored in *destination*. This result can occur if *max_bytes* is zero.
-1	An encoding error has occurred. The next *max_bytes* or fewer bytes do not contribute to a complete and valid multibyte character. No value is stored in *destination*. **EILSEQ** is stored in **errno** and the conversion state *state* is unspecified.

Remarks

The **mbrtoc16** function reads up to *max_bytes* bytes from *source* to find the first complete, valid multibyte character, and then stores the equivalent UTF-16 character in *destination*. The source bytes are interpreted according to the current thread multibyte locale. If the multibyte character requires more than one UTF-16 output character, such as a surrogate pair, then the *state* value is set to store the next UTF-16 character in *destination* on the next call to **mbrtoc16**. The **mbrtoc32** function is identical, but output is stored as a UTF-32 character.

If *source* is null, these functions return the equivalent of a call made using arguments of **NULL** for *destination*, **""** for *source*, and 1 for *max_bytes*. The passed values of *destination* and *max_bytes* are ignored.

If *source* is not null, the function starts at the beginning of the string and inspects up to *max_bytes* bytes to determine the number of bytes required to complete the next multibyte character, including any shift sequences. If the examined bytes contain a valid and complete multibyte character, the function converts the character into the equivalent 16-bit or 32-bit wide character or characters. If *destination* is not null, the function stores the first (and possibly only) result character in destination. If additional output characters are required, a value is set in *state*, so that subsequent calls to the function output the additional characters and return the value -3. If no more output characters are required, then *state* is set to the initial shift state.

Requirements

FUNCTION	C HEADER	C++ HEADER
mbrtoc16, **mbrtoc32**	<uchar.h>	<cuchar>

For additional compatibility information, see Compatibility.

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
c16rtomb, c32rtomb

mbrtowc
mbsrtowcs
mbsrtowcs_s

mbrtowc

10/31/2018 • 3 minutes to read • Edit Online

Convert a multibyte character in the current locale into the equivalent wide character, with the capability of restarting in the middle of a multibyte character.

Syntax

```
size_t mbrtowc(
   wchar_t *wchar,
   const char *mbchar,
   size_t count,
   mbstate_t *mbstate
);
```

Parameters

wchar
Address of a wide character to receive the converted wide character string (type **wchar_t**). This value can be a null pointer if no return wide character is required.

mbchar
Address of a sequence of bytes (a multibyte character).

count
Number of bytes to check.

mbstate
Pointer to conversion state object. If this value is a null pointer, the function uses a static internal conversion state object. Because the internal **mbstate_t** object is not thread-safe, we recommend that you always pass your own *mbstate* argument.

Return Value

One of the following values:

0 The next *count* or fewer bytes complete the multibyte character that represents the null wide character, which is stored in *wchar*, if *wchar* is not a null pointer.

1 to *count*, inclusive The next *count* or fewer bytes complete a valid multibyte character. The value returned is the number of bytes that complete the multibyte character. The wide character equivalent is stored in *wchar*, if *wchar* is not a null pointer.

(size_t)(-1) An encoding error occurred. The next *count* or fewer bytes do not contribute to a complete and valid multibyte character. In this case, **errno** is set to EILSEQ and the conversion shift state in *mbstate* is unspecified.

(size_t)(-2) The next *count* bytes contribute to an incomplete but potentially valid multibyte character, and all *count* bytes have been processed. No value is stored in *wchar*, but *mbstate* is updated to restart the function.

Remarks

If *mbchar* is a null pointer, the function is equivalent to the call:

```
mbrtowc(NULL, "", 1, &mbstate)
```

In this case, the value of the arguments *wchar* and *count* are ignored.

If *mbchar* is not a null pointer, the function examines *count* bytes from *mbchar* to determine the required number of bytes that are required to complete the next multibyte character. If the next character is valid, the corresponding multibyte character is stored in *wchar* if it is not a null pointer. If the character is the corresponding wide null character, the resulting state of *mbstate* is the initial conversion state.

The **mbrtowc** function differs from mbtowc, _mbtowc_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application should use **wcsrlen** instead of **wcslen** if a subsequent call to **wcsrtombs** is used instead of **wcstombs**.

Example

Converts a multibyte character to its wide character equivalent.

```
// crt_mbrtowc.cpp

#include <stdio.h>
#include <mbctype.h>
#include <string.h>
#include <locale.h>
#include <wchar.h>

#define BUF_SIZE 100

int Sample(char* szIn, wchar_t* wcOut, int nMax)
{
    mbstate_t   state = {0}; // Initial state
    size_t      nConvResult,
                nmbLen = 0,
                nwcLen = 0;
    wchar_t*    wcCur = wcOut;
    wchar_t*    wcEnd = wcCur + nMax;
    const char* mbCur = szIn;
    const char* mbEnd = mbCur + strlen(mbCur) + 1;
    char*       szLocal;

    // Sets all locale to French_Canada.1252
    szLocal = setlocale(LC_ALL, "French_Canada.1252");
    if (!szLocal)
    {
        printf("The fuction setlocale(LC_ALL, \"French_Canada.1252\") failed!\n");
        return 1;
    }

    printf("Locale set to: \"%s\"\n", szLocal);

    // Sets the code page associated current locale's code page
    // from a previous call to setlocale.
    if (_setmbcp(_MB_CP_SBCS) == -1)
    {
        printf("The fuction _setmbcp(_MB_CP_SBCS) failed!");
        return 1;
    }

    while ((mbCur < mbEnd) && (wcCur < wcEnd))
    {
        //
        nConvResult = mbrtowc(wcCur, mbCur, 1, &state);
        switch (nConvResult)
        {
            case 0:
```

```
            case 0:
            {  // done
                printf("Conversion succeeded!\nMultibyte String: ");
                printf(szIn);
                printf("\nWC String: ");
                wprintf(wcOut);
                printf("\n");
                mbCur = mbEnd;
                break;
            }

            case -1:
            {  // encoding error
                printf("The call to mbrtowc has detected an encoding error.\n");
                mbCur = mbEnd;
                break;
            }

            case -2:
            {  // incomplete character
                if   (!mbsinit(&state))
                {
                    printf("Currently in middle of mb conversion, state = %x\n", state);
                    // state will contain data regarding lead byte of mb character
                }

                ++nmbLen;
                ++mbCur;
                break;
            }

            default:
            {
                if   (nConvResult > 2) // The multibyte should never be larger than 2
                {
                    printf("Error: The size of the converted multibyte is %d.\n", nConvResult);
                }

                ++nmbLen;
                ++nwcLen;
                ++wcCur;
                ++mbCur;
            break;
            }
        }
    }

   return 0;
}

int main(int argc, char* argv[])
{
    char    mbBuf[BUF_SIZE] = "AaBbCc\x9A\x8B\xE0\xEF\xF0xXyYzZ";
    wchar_t wcBuf[BUF_SIZE] = {L''};

    return Sample(mbBuf, wcBuf, BUF_SIZE);
}
```

Sample Output

```
Locale set to: "French_Canada.1252"
Conversion succeeded!
Multibyte String: AaBbCcÜïαn≡xXyYzZ
WC String: AaBbCcÜïαn≡xXyYzZ
```

Requirements

ROUTINE	REQUIRED HEADER
mbrtowc	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences

_mbsbtype, _mbsbtype_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the type of byte within a string.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _mbsbtype(
   const unsigned char *mbstr,
   size_t count
);
int _mbsbtype_l(
   const unsigned char *mbstr,
   size_t count,
   _locale_t locale
);
```

Parameters

mbstr
Address of a sequence of multibyte characters.

count
Byte offset from head of string.

locale
Locale to use.

Return Value

_mbsbtype and **_mbsbtype_l** returns an integer value indicating the result of the test on the specified byte. The manifest constants in the following table are defined in Mbctype.h.

RETURN VALUE	BYTE TYPE
_MBC_SINGLE (0)	Single-byte character. For example, in code page 932, **_mbsbtype** returns 0 if the specified byte is within the range 0x20 - 0x7E or 0xA1 - 0xDF.
_MBC_LEAD (1)	Lead byte of multibyte character. For example, in code page 932, **_mbsbtype** returns 1 if the specified byte is within the range 0x81 - 0x9F or 0xE0 - 0xFC.
_MBC_TRAIL (2)	Trailing byte of multibyte character. For example, in code page 932, **_mbsbtype** returns 2 if the specified byte is within the range 0x40 - 0x7E or 0x80 - 0xFC.

RETURN VALUE	BYTE TYPE
_MBC_ILLEGAL (-1)	**NULL** string, invalid character, or null byte found before the byte at offset *count* in *mbstr*.

Remarks

The **_mbsbtype** function determines the type of a byte in a multibyte character string. The function examines only the byte at offset *count* in *mbstr*, ignoring invalid characters before the specified byte.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The version of this function without the **_l** suffix uses the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it use the locale parameter passed in instead. For more information, see Locale.

If the input string is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **_MBC_ILLEGAL**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_mbsbtype	<mbstring.h>	<mbctype.h>*
_mbsbtype_l	<mbstring.h>	<mbctype.h>*

* For manifest constants used as return values.

For more compatibility information, see Compatibility.

See also

Byte Classification

mbsinit

10/31/2018 • 2 minutes to read • Edit Online

Tracks the state of a multibyte character conversion.

Syntax

```
int mbsinit(
   const mbstate_t* ps
);
```

Parameters

ps
A pointer to an mbstate_t variable.

Return Value

Nonzero if *ps* is **NULL** or if not in the middle of a conversion.

Remarks

When using one of the ANSI functions that takes an **mbstate_t** pointer, passing the address of your **mbstate_t** will return information about whether the last byte in the buffer was converted.

The appropriate code page needs to be installed to support your multibyte characters.

Example

```
// crt_mbsinit.cpp
#include <stdio.h>
#include <mbctype.h>
#include <string.h>
#include <locale.h>
#include <cwchar>
#include <xlocinfo.h>

#define   BUF_SIZE   0x40

wchar_t      g_wcBuf[BUF_SIZE] = L"This a wc buffer that will be over written...";
char      g_mbBuf[BUF_SIZE] = "AaBbCc\x9A\x8B\xE0\xEF\xF0xXyYzZ";
int      g_nInit = strlen(g_mbBuf);

int MbsinitSample(char* szIn, wchar_t* wcOut, int nMax)
{
   mbstate_t   state = {0};
   size_t      nConvResult, nmbLen = 0, nwcLen = 0;
   wchar_t*   wcCur = wcOut;
   wchar_t*   wcEnd = wcCur + nMax;
   const char*   mbCur = szIn;
   const char*   mbEnd = mbCur + strlen(mbCur) + 1;
   char*      szLocal = setlocale(LC_ALL, "japanese");

   printf("Locale set to: \"%s\"\n", szLocal);

   if   (_setmbcp(_MB_CP_LOCALE) != -1)
```

```
   {
      while   ((mbCur < mbEnd) && (wcCur < wcEnd))
      {
         nConvResult = mbrtowc(wcCur, mbCur, 1, &state);

         switch   (nConvResult)
         {
            case 0:
            {   // done
               printf("Conversion succeeded!\nMB String: ");
               printf(szIn);
               printf("\nWC String: ");
               wprintf(wcOut);
               printf("\n");
               mbCur = mbEnd;
               break;
            }

            case -1:
            {   // encoding error
               printf("ERROR: The call to mbrtowc has detected an encoding error.\n");
               mbCur = mbEnd;
               break;
            }

            case -2:
            {   // incomplete character
               if   (!mbsinit(&state))
               {
                  printf("Currently in middle of mb conversion, state = %x\n", state);
                  // state will contain data regarding lead byte of mb character
               }

               ++nmbLen;
               ++mbCur;
               break;
            }

            default:
            {
               if   (nConvResult > 2)   // Microsoft mb should never be larger than 2
                  printf("ERROR: nConvResult = %d\n", nConvResult);

               ++nmbLen;
               ++nwcLen;
               ++wcCur;
               ++mbCur;
               break;
            }
         }
      }
   }
   else
      printf("ERROR: _setmbcp(932) failed!");

   return 0;
}

int main(int argc, char* argv[])
{
   return MbsinitSample(g_mbBuf, g_wcBuf, BUF_SIZE);
}
```

Sample Output

```
Locale set to: "Japanese_Japan.932"
Currently in middle of mb conversion, state = 9a
Currently in middle of mb conversion, state = e0
Currently in middle of mb conversion, state = f0
Conversion succeeded!
MB String: AaBbCcxXyYzZ
WC String: AaBbCcxXyYzZ
```

See also

Byte Classification

_mbsnbcat, _mbsnbcat_l

10/31/2018 • 2 minutes to read • Edit Online

Appends, at most, the first **n** bytes of one multibyte-character string to another. More secure versions of these functions are available; see _mbsnbcat_s, _mbsnbcat_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned char *_mbsnbcat(
   unsigned char *dest,
   const unsigned char *src,
   size_t count
);
unsigned char *_mbsnbcat_l(
   unsigned char *dest,
   const unsigned char *src,
   size_t count,
   _locale_t locale
);
template <size_t size>
unsigned char *_mbsnbcat(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsnbcat_l(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

dest
Null-terminated multibyte-character destination string.

src
Null-terminated multibyte-character source string.

count
Number of bytes from *src* to append to *dest*.

locale
Locale to use.

Return Value

_mbsnbcat returns a pointer to the destination string. No return value is reserved to indicate an error.

Remarks

The **_mbsnbcat** function appends, at most, the first *count* bytes of *src* to *dest*. If the byte immediately preceding the null character in *dest* is a lead byte, the initial byte of *src* overwrites this lead byte. Otherwise, the initial byte of *src* overwrites the terminating null character of *dest*. If a null byte appears in *src* before *count* bytes are appended, **_mbsnbcat** appends all bytes from *src*, up to the null character. If *count* is greater than the length of *src*, the length of *src* is used in place of *count*. The resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbsnbcat** version of the function uses the current locale for this locale-dependent behavior; the **_mbsnbcat_l** version is identical except that they use the locale parameter passed in instead. For more information, see Locale.

Security Note Use a null-terminated string. The null-terminated string must not exceed the size of the destination buffer. For more information, see Avoiding Buffer Overruns.

If *dest* or *src* is **NULL**, the function will generate an invalid parameter error, as described in Parameter Validation. If the error is handled, the function returns **EINVAL** and sets **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncat	strncat	**_mbsnbcat**	wcsncat
_tcsncat_l	**_strncat_l**	**_mbsnbcat_l**	**_wcsncat_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcat	<mbstring.h>
_mbsnbcat_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_mbsnbcmp, _mbsnbcmp_l
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l
_mbsnbcpy, _mbsnbcpy_l
_mbsnbicmp, _mbsnbicmp_l
_mbsnbset, _mbsnbset_l
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
_mbsnbcat_s, _mbsnbcat_s_l

_mbsnbcat_s, _mbsnbcat_s_l

10/31/2018 • 2 minutes to read • Edit Online

Appends to a multibyte character string, at most, the first **n** bytes of another multibyte-character string. These are versions of _mbsnbcat, _mbsnbcat_l that have security enhancements, as described in Security Features in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _mbsnbcat_s(
   unsigned char *dest,
   size_t sizeInBytes,
   const unsigned char *src,
   size_t count
);
errno_t _mbsnbcat_s_l(
   unsigned char *dest,
   size_t sizeInBytes,
   const unsigned char *src,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbcat_s(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbcat_s_l(
   unsigned char (&dest)[size],
   const unsigned char *src,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

dest
Null-terminated multibyte-character destination string.

sizeInBytes
Size of the *dest* buffer in bytes.

src
Null-terminated multibyte-character source string.

count
Number of bytes from *src* to append to *dest*.

locale

Locale to use.

Return Value

Zero if successful; otherwise, an error code.

Error Conditions

DEST	*SIZEINBYTES*	*SRC*	RETURN VALUE
NULL	any	any	**EINVAL**
Any	<= 0	any	**EINVAL**
Any	any	**NULL**	**EINVAL**

If any of the error conditions occurs, the function generates an invalid parameter error, as described in Parameter Validation. If the error is handled, the function returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **_mbsnbcat_s** function appends to *dest*, at most, the first *count* bytes of *src*. If the byte that immediately precedes the null character in *dest* is a lead byte, it is overwritten by the initial byte of *src*. Otherwise, the initial byte of *src* overwrites the terminating null character of *dest*. If a null byte appears in *src* before *count* bytes are appended, **_mbsnbcat_s** appends all bytes from *src*, up to the null character. If *count* is greater than the length of *src*, the length of *src* is used in place of *count*. The resulting string is terminated by a null character. If copying takes place between strings that overlap, the behavior is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions are identical, except that the ones that don't have the **_l** suffix use the current locale and the ones that do have the **_l** suffix instead use the locale parameter that's passed in. For more information, see Locale.

In C++, the use of these functions is simplified by template overloads; the overloads can infer buffer length automatically and thereby eliminate the need to specify a size argument, and they can automatically use their newer, more secure functions to replace older, less-secure functions. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncat	strncat	**_mbsnbcat_s**	wcsncat
_tcsncat_s_l	**_strncat_s_l**	**_mbsnbcat_s_l**	**_wcsncat_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcat_s	<mbstring.h>

ROUTINE	REQUIRED HEADER
_mbsnbcat_s_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_mbsnbcmp, _mbsnbcmp_l
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l
_mbsnbcpy, _mbsnbcpy_l
_mbsnbcpy_s, _mbsnbcpy_s_l
_mbsnbset, _mbsnbset_l
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

_mbsnbcmp, _mbsnbcmp_l

10/31/2018 • 2 minutes to read • Edit Online

Compares the first **n** bytes of two multibyte-character strings.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _mbsnbcmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbcmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*
The strings to compare.

count
The number of bytes to compare.

locale
The locale to use.

Return Value

The return value indicates the ordinal relationship between the substrings of *string1* and *string2*.

RETURN VALUE	DESCRIPTION
< 0	*string1* substring is less than *string2* substring.
0	*string1* substring is identical to *string2* substring.
> 0	*string1* substring is greater than *string2* substring.

On a parameter validation error, **_mbsnbcmp** and **_mbsnbcmp_l** return **_NLSCMPERROR**, which is defined in <string.h> and <mbstring.h>.

Remarks

The **_mbsnbcmp** functions compare at most the first *count* bytes in *string1* and *string2* and return a value that indicates the relationship between the substrings. **_mbsnbcmp** is a case-sensitive version of **_mbsnbicmp**. Unlike **_mbsnbcoll**, **_mbsnbcmp** is not affected by the collation order of the locale. **_mbsnbcmp** recognizes multibyte-character sequences according to the current multibyte code page.

_mbsnbcmp resembles **_mbsncmp**, except that **_mbsncmp** compares strings by characters rather than by bytes.

The output value is affected by the **LC_CTYPE** category setting of the locale, which specifies the lead bytes and trailing bytes of multibyte characters. For more information, see setlocale. The **_mbsnbcmp** function uses the current locale for this locale-dependent behavior. The **_mbsnbcmp_l** function is identical except that it uses the *locale* parameter instead. For more information, see Locale.

If either *string1* or *string2* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the functions return **_NLSCMPERROR** and **errno** is set to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncmp	strncmp	**_mbsnbcmp**	wcsncmp
_tcsncmp_l	strncmp	**_mbsnbcml**	wcsncmp

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcmp	<mbstring.h>
_mbsnbcmp_l	<mbstring.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mbsnbcmp.c
#include <mbstring.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

int main( void )
{
   char tmp[20];
   int result;
   printf( "Compare strings:\n          %s\n", string1 );
   printf( "          %s\n\n", string2 );
   printf( "Function: _mbsnbcmp (first 10 characters only)\n" );
   result = _mbsncmp( string1, string2 , 10 );
   if( result > 0 )
      _mbscpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      _mbscpy_s( tmp, sizeof(tmp), "less than" );
   else
      _mbscpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:   String 1 is %s string 2\n\n", tmp );
   printf( "Function: _mbsnicmp _mbsnicmp (first 10 characters only)\n" );
   result = _mbsnicmp( string1, string2, 10 );
   if( result > 0 )
      _mbscpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      _mbscpy_s( tmp, sizeof(tmp), "less than" );
   else
      _mbscpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:   String 1 is %s string 2\n\n", tmp );
}
```

Output

```
Compare strings:
          The quick brown dog jumps over the lazy fox
          The QUICK brown fox jumps over the lazy dog

Function: _mbsnbcmp (first 10 characters only)
Result:   String 1 is greater than string 2

Function: _mbsnicmp _mbsnicmp (first 10 characters only)
Result:   String 1 is equal to string 2
```

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_mbsnbicmp, _mbsnbicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
Locale
Interpretation of Multibyte-Character Sequences

_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l

10/31/2018 • 3 minutes to read • Edit Online

Compares *n* bytes of two multibyte-character strings by using multibyte code-page information.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _mbsnbcoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbcoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
int _mbsnbicoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsnbicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*
Strings to compare.

count
Number of bytes to compare.

locale
Locale to use.

Return Value

The return value indicates the relation of the substrings of *string1* and *string2*.

RETURN VALUE	DESCRIPTION
< 0	*string1* substring less than *string2* substring.

RETURN VALUE	DESCRIPTION
0	*string1* substring identical to *string2* substring.
> 0	*string1* substring greater than *string2* substring.

If *string1* or *string2* is **NULL** or *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**. To use **_NLSCMPERROR**, include either String.h or Mbstring.h.

Remarks

Each of these functions collates, at most, the first *count* bytes in *string1* and *string2* and returns a value indicating the relationship between the resulting substrings of *string1* and *string2*. If the final byte in the substring of *string1* or *string2* is a lead byte, it is not included in the comparison; these functions compare only complete characters in the substrings. **_mbsnbicoll** is a case-insensitive version of **_mbsnbcoll**. Like _mbsnbcmp and _mbsnbicmp, **_mbsnbcoll** and **_mbsnbicoll** collate the two multibyte-character strings according to the lexicographic order specified by the multibyte code page currently in use.

For some code pages and corresponding character sets, the order of characters in the character set might differ from the lexicographic character order. In the "C" locale, this is not the case: the order of characters in the ASCII character set is the same as the lexicographic order of the characters. However, in certain European code pages, for example, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically. To perform a lexicographic comparison of strings by bytes in such an instance, use **_mbsnbcoll** rather than **_mbsnbcmp**; to check only for string equality, use **_mbsnbcmp**.

Because the **coll** functions collate strings lexicographically for comparison, whereas the **cmp** functions simply test for string equality, the **coll** functions are much slower than the corresponding **cmp** versions. Therefore, the **coll** functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the comparison.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncoll	_strncoll	**_mbsnbcoll**	_wcsncoll
_tcsncoll_l	_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l	**_mbsnbcoll_l**	_wcsncoll_l
_tcsnicoll	_strnicoll	**_mbsnbicoll**	_wcsnicoll
_tcsnicoll_l	_strnicoll_l	**_mbsnbicoll_l**	_wcsnicoll_l

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcoll	<mbstring.h>
_mbsnbcoll_l	<mbstring.h>
_mbsnbicoll	<mbstring.h>
_mbsnbicoll_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_mbsnbicmp, _mbsnbicmp_l
strcoll Functions
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

_mbsnbcpy, _mbsnbcpy_l

10/31/2018 • 2 minutes to read • Edit Online

Copies **n** bytes of a string to a destination string. More secure versions of these functions are available—see _mbsnbcpy_s, _mbsnbcpy_s_l.

> **IMPORTANT**
>
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned char * _mbsnbcpy(
   unsigned char * strDest,
   const unsigned char * strSource,
   size_t count
);
unsigned char * _mbsnbcpy_l(
   unsigned char * strDest,
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
unsigned char * _mbsnbcpy(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char * _mbsnbcpy_l(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest
Destination for the character string to be copied.

strSource
Character string to be copied.

count
Number of bytes to be copied.

locale
Locale to use.

Return Value

_mbsnbcpy returns a pointer to the destination character string. No return value is reserved to indicate an error.

Remarks

The **_mbsnbcpy** function copies *count* bytes from *strSource* to *strDest*. If *count* exceeds the size of *strDest* or the source and destination strings overlap, the behavior of **_mbsnbcpy** is undefined.

If *strSource* or *strDest* is a null pointer, this function invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions are identical, except that those that don't have the **_l** suffix use the current locale and the versions that do have the **_l** suffix instead use the locale parameter that's passed in. For more information, see Locale.

> **IMPORTANT**
> These functions might be vulnerable to buffer overrun threats. Buffer overruns can be used to execute arbitrary attacker code, which can cause an unwarranted elevation of privilege and compromise the system. For more information, see Avoiding Buffer Overruns.

In C++, these functions have template overloads that invoke the newer, more secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncpy	strncpy	**_mbsnbcpy**	wcsncpy
_tcsncpy_l	**_strncpy_l**	**_mbsnbcp_l**	**_wcsncpy_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcpy	<mbstring.h>
_mbsnbcpy_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l
_mbsnbset, _mbsnbset_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

_mbsnbcpy_s, _mbsnbcpy_s_l

10/31/2018 • 2 minutes to read • Edit Online

Copies **n** bytes of a string to a destination string. These versions of _mbsnbcpy, _mbsnbcpy_l have security enhancements, as described in Security Features in the CRT.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _mbsnbcpy_s(
   unsigned char * strDest,
   size_t sizeInBytes,
   const unsigned char * strSource,
   size_t count
);
errno_t _mbsnbcpy_s_l(
   unsigned char * strDest,
   size_t sizeInBytes,
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbcpy_s(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbcpy_s_l(
   unsigned char (&strDest)[size],
   const unsigned char * strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest
Destination for character string to be copied.

sizeInBytes
Destination buffer size.

strSource
Character string to be copied.

count
Number of bytes to be copied.

locale
Locale to use.

Return Value

Zero if successful; **EINVAL** if a bad parameter was passed in.

Remarks

The **_mbsnbcpy_s** function copies *count* bytes from *strSource* to *strDest*. If *count* exceeds the size of *strDest*, either of the input strings is a null pointer, or *sizeInBytes* or *count* is 0, the function invokes the invalid parameter handler as described in Parameter Validation . If execution is allowed to continue, the function returns **EINVAL**. If the source and destination strings overlap, the behavior of **_mbsnbcpy_s** is undefined.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

> **NOTE**
> Unlike the non-secure version of this function, **_mbsnbcpy_s** does not do any null padding and always null terminates the string.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncpy_s	**_strncpy_s**	**_mbsnbcpy_s**	**_wcsncpy_s**
_tcsncpy_s_l	**_strncpy_s_l**	**_mbsnbcpy_s_l**	**_wcsncpy_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcpy_s	<mbstring.h>
_mbsnbcpy_s_l	<mbstring.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l
_mbsnbicmp, _mbsnbicmp_l

_mbsnbset, _mbsnbset_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

_mbsnbicmp, _mbsnbicmp_l

10/31/2018 • 2 minutes to read • Edit Online

Compares **n** bytes of two multibyte-character strings, and ignores case.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _mbsnbicmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

count
Number of bytes to compare.

Return Value

The return value indicates the relationship between the substrings.

RETURN VALUE	DESCRIPTION
< 0	*string1* substring less than *string2* substring.
0	*string1* substring identical to *string2* substring.
> 0	*string1* substring greater than *string2* substring.

On an error, **_mbsnbicmp** returns **_NLSCMPERROR**, which is defined in String.h and Mbstring.h.

Remarks

The **_mbsnbicmp** function performs an ordinal comparison of at most the first *count* bytes of *string1* and *string2*. The comparison is performed by converting each character to lowercase; _mbsnbcmp is a case-sensitive version of **_mbsnbicmp**. The comparison ends if a terminating null character is reached in either string before *count* characters are compared. If the strings are equal when a terminating null character is reached in either string before *count* characters are compared, the shorter string is lesser.

_mbsnbicmp is similar to _mbsnbcmp, except that it compares strings up to *count* bytes instead of by characters.

Two strings containing characters located between 'Z' and 'a' in the ASCII table ('[', '\', ']', '^', '_', and '`') compare differently, depending on their case. For example, the two strings "ABCDE" and "ABCD^" compare one way if the comparison is lowercase ("abcde" > "abcd^") and the other way ("ABCDE" < "ABCD^") if it is uppercase.

_mbsnbicmp recognizes multibyte-character sequences according to the multibyte code page currently in use. It is not affected by the current locale setting.

If either *string1* or *string2* is a null pointer, **_mbsnbicmp** invokes the invalid parameter handler as described in Parameter Validation. If execution is allowed to continue, the function returns **_NLSCMPERROR** and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnicmp	**_strnicmp**	**_mbsnbicmp**	**_wcsnicmp**
_tcsnicmp_l	**_strnicmp_l**	**_mbsnbicmp_l**	**_wcsnicmp_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbicmp	<mbstring.h>

For more compatibility information, see Compatibility.

Example

See the example for _mbsnbcmp, _mbsnbcmp_l.

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_mbsnbcmp, _mbsnbcmp_l
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

_mbsnbset, _mbsnbset_l

10/31/2018 • 2 minutes to read • Edit Online

Sets the first **n** bytes of a multibyte-character string to a specified character. More secure versions of these functions are available; see _mbsnbset_s, _mbsnbset_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned char *_mbsnbset(
   unsigned char *str,
   unsigned int c,
   size_t count
);
unsigned char *_mbsnbset_l(
   unsigned char *str,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str
String to be altered.

c
Single-byte or multibyte-character setting.

count
Number of bytes to be set.

locale
Locale to use.

Return Value

_mbsnbset returns a pointer to the altered string.

Remarks

The **_mbsnbset** and **_mbsnbset_l** functions set, at most, the first *count* bytes of *str* to *c*. If *count* is greater than the length of *str*, the length of *str* is used instead of *count*. If *c* is a multibyte character and cannot be set entirely into the last byte specified by *count*, the last byte is padded with a blank character. **_mbsnbset** and **_mbsnbset_l** does not place a terminating null at the end of *str*.

_mbsnbset and **_mbsnbset_l** is similar to **_mbsnset**, except that it sets *count* bytes rather than *count* characters of *c*.

If *str* is **NULL** or *count* is zero, this function generates an invalid parameter exception as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. Also, if *c* is not a valid multibyte character, **errno** is set to **EINVAL** and a space is used instead.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The **_mbsnbset** version of this function uses the current locale for this locale-dependent behavior; the **_mbsnbset_l** version is identical except that it use the locale parameter passed in instead. For more information, see Locale.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnset	**_strnset**	**_mbsnbset**	**_wcsnset**
_tcsnset_l	**_strnset_l**	**_mbsnbset_l**	**_wcsnset_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbset	<mbstring.h>
_mbsnbset_l	<mbstring.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mbsnbset.c
// compile with: /W3
#include <mbstring.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 bytes of string to be *'s */
   printf( "Before: %s\n", string );
   _mbsnbset( string, '*', 4 ); // C4996
   // Note; _mbsnbset is deprecated; consider _mbsnbset_s
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

See also

String Manipulation

_mbsnbcat, _mbsnbcat_l

_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_mbsnbset_s, _mbsnbset_s_l

10/31/2018 • 2 minutes to read • Edit Online

Sets the first **n** bytes of a multibyte-character string to a specified character. These versions of _mbsnbset, _mbsnbset_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _mbsnbset_s(
   unsigned char *str,
   size_t size,
   unsigned int c,
   size_t count
);
errno_t _mbsnbset_s_l(
   unsigned char *str,
   size_t size,
   unsigned int c,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t _mbsnbset_s(
   unsigned char (&str)[size],
   unsigned int c,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsnbset_s_l(
   unsigned char (&str)[size],
   unsigned int c,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

str
String to be altered.

size
The size of the string buffer.

c
Single-byte or multibyte-character setting.

count
Number of bytes to be set.

locale
Locale to use.

Return Value

Zero if successful; otherwise, an error code.

Remarks

The **_mbsnbset_s** and **_mbsnbset_s_l** functions set, at most, the first *count* bytes of *str* to *c*. If *count* is greater than the length of *str*, the length of *str* is used instead of *count*. If *c* is a multibyte character and cannot be set entirely into the last byte that's specified by *count*, the last byte is padded with a blank character. **_mbsnbset_s** and **_mbsnbset_s_l** do not place a terminating null at the end of *str*.

_mbsnbset_s and **_mbsnbset_s_l** resemble **_mbsnset**, except that they set *count* bytes rather than *count* characters of *c*.

If *str* is **NULL** or *count* is zero, this function generates an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **NULL**. Also, if *c* is not a valid multibyte character, **errno** is set to **EINVAL** and a space is used instead.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The **_mbsnbset_s** version of this function uses the current locale for this locale-dependent behavior; the **_mbsnbset_s_l** version is identical except that it instead uses the locale parameter that's passed in. For more information, see Locale.

In C++, use of these functions is simplified by template overloads; the overloads can infer buffer length automatically and thereby eliminate the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnset_s	**_strnset_s**	**_mbsnbset_s**	**_wcsnset_s**
_tcsnset_s_l	`_strnset_s _l`	**_mbsnbset_s_l**	**_wcsnset_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbset_s	<mbstring.h>
_mbsnbset_s_l	<mbstring.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mbsnbset_s.c
#include <mbstring.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 bytes of string to be *'s */
   printf( "Before: %s\n", string );
   _mbsnbset_s( string, sizeof(string), '*', 4 );
   printf( "After:  %s\n", string );
}
```

Output

```
Before: This is a test
After:  **** is a test
```

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

mbsrtowcs

10/31/2018 • 3 minutes to read • Edit Online

Converts a multibyte character string in the current locale to a corresponding wide character string, with the capability of restarting in the middle of a multibyte character. A more secure version of this function is available; see mbsrtowcs_s.

Syntax

```
size_t mbsrtowcs(
   wchar_t *wcstr,
   const char **mbstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
size_t mbsrtowcs(
   wchar_t (&wcstr)[size],
   const char **mbstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

wcstr
Address to store the resulting converted wide character string.

mbstr
Indirect pointer to the location of the multibyte character string to convert.

count
The maximum number of characters (not bytes) to convert and store in *wcstr*.

mbstate
A pointer to an **mbstate_t** conversion state object. If this value is a null pointer, a static internal conversion state object is used. Because the internal **mbstate_t** object is not thread-safe, we recommend that you always pass your own *mbstate* parameter.

Return Value

Returns the number of characters successfully converted, not including the terminating null character, if any. Returns (size_t)(-1) if an error occurred, and sets **errno** to EILSEQ.

Remarks

The **mbsrtowcs** function converts a string of multibyte characters indirectly pointed to by *mbstr*, into wide characters stored in the buffer pointed to by *wcstr*, by using the conversion state contained in *mbstate*. The conversion continues for each character until either a terminating null multibyte character is encountered, a multibyte sequence that does not correspond to a valid character in the current locale is encountered, or until *count* characters have been converted. If **mbsrtowcs** encounters the multibyte null character ('\0') either before or when *count* occurs, it converts it to a 16-bit terminating null character and stops.

Thus, the wide character string at *wcstr* is null-terminated only if **mbsrtowcs** encounters a multibyte null character

during conversion. If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbsrtowcs** is undefined. **mbsrtowcs** is affected by the LC_TYPE category of the current locale.

The **mbsrtowcs** function differs from mbstowcs, _mbstowcs_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application should use **mbsrlen** instead of **mbslen**, if a subsequent call to **mbsrtowcs** is used instead of **mbstowcs**.

If *wcstr* is not a null pointer, the pointer object pointed to by *mbstr* is assigned a null pointer if conversion stopped because a terminating null character was reached. Otherwise, it is assigned the address just past the last multibyte character converted, if any. This allows a subsequent function call to restart conversion where this call stopped.

If the *wcstr* argument is a null pointer, the *count* argument is ignored and **mbsrtowcs** returns the required size in wide characters for the destination string. If *mbstate* is a null pointer, the function uses a non-thread-safe static internal **mbstate_t** conversion state object. If the character sequence *mbstr* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

If *mbstr* isa null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns -1.

In C++, this function has a template overload that invokes the newer, secure counterpart of this function. For more information, see Secure Template Overloads.

Exceptions

The **mbsrtowcs** function is multithread safe as long as no function in the current thread calls **setlocale** as long as this function is executing and the *mbstate* argument is not a null pointer.

Requirements

ROUTINE	REQUIRED HEADER
mbsrtowcs	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbrtowc
mbtowc, _mbtowc_l
mbstowcs, _mbstowcs_l
mbsinit

mbsrtowcs_s

10/31/2018 • 4 minutes to read • Edit Online

Convert a multibyte character string in the current locale to its wide character string representation. A version of mbsrtowcs with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t mbsrtowcs_s(
   size_t * pReturnValue,
   wchar_t * wcstr,
   size_t sizeInWords,
   const char ** mbstr,
   size_t count,
   mbstate_t * mbstate
);
template <size_t size>
errno_t mbsrtowcs_s(
   size_t * pReturnValue,
   wchar_t (&wcstr)[size],
   const char ** mbstr,
   size_t count,
   mbstate_t * mbstate
); // C++ only
```

Parameters

pReturnValue
The number of characters converted.

wcstr
Address of buffer to store the resulting converted wide character string.

sizeInWords
The size of *wcstr* in words (wide characters).

mbstr
Indirect pointer to the location of the multibyte character string to be converted.

count
The maximum number of wide characters to store in the *wcstr* buffer, not including the terminating null, or _TRUNCATE.

mbstate
A pointer to an **mbstate_t** conversion state object. If this value is a null pointer, a static internal conversion state object is used. Because the internal **mbstate_t** object is not thread-safe, we recommend that you always pass your own *mbstate* parameter.

Return Value

Zero if conversion is successful, or an error code on failure.

ERROR CONDITION	RETURN VALUE AND ERRNO
wcstr is a null pointer and *sizeInWords* > 0	**EINVAL**
mbstr is a null pointer	**EINVAL**
The string indirectly pointed to by *mbstr* contains a multibyte sequence that is not valid for the current locale.	**EILSEQ**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; for more information, see Remarks)	**ERANGE**

If any one of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **mbsrtowcs_s** function converts a string of multibyte characters indirectly pointed to by *mbstr* into wide characters stored in the buffer pointed to by *wcstr*, by using the conversion state contained in *mbstate*. The conversion will continue for each character until one of these conditions is met:

- A multibyte null character is encountered
- An invalid multibyte character is encountered
- The number of wide characters stored in the *wcstr* buffer equals *count*.

The destination string *wcstr* is always null-terminated, even in the case of an error, unless *wcstr* is a null pointer.

If *count* is the special value _TRUNCATE, **mbsrtowcs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **mbsrtowcs_s** successfully converts the source string, it puts the size in wide characters of the converted string and the null terminator into **pReturnValue*, provided *pReturnValue* is not a null pointer. This occurs even if the *wcstr* argument is a null pointer and lets you determine the required buffer size. Note that if *wcstr* is a null pointer, *count* is ignored.

If *wcstr* is not a null pointer, the pointer object pointed to by *mbstr* is assigned a null pointer if conversion stopped because a terminating null character was reached. Otherwise, it is assigned the address just past the last multibyte character converted, if any. This allows a subsequent function call to restart conversion where this call stopped.

If *mbstate* is a null pointer, the library internal **mbstate_t** conversion state static object is used. Because this internal static object is not thread-safe, we recommend that you pass your own *mbstate* value.

If **mbsrtowcs_s** encounters a multibyte character that is not valid in the current locale, it puts -1 in **pReturnValue*, sets the destination buffer *wcstr* to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbsrtowcs_s** is undefined. **mbsrtowcs_s** is affected by the LC_TYPE category of the current locale.

> **IMPORTANT**
>
> Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

The **mbsrtowcs_s** function differs from mbstowcs_s, _mbstowcs_s_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application should use **mbsrlen** instead of **mbslen**, if a subsequent call to **mbsrtowcs_s** is used instead of **mbstowcs_s**.

In C++, using this function is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the requirement to specify a size argument) and they can automatically replace older, non-secure functions by using their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **mbsrtowcs_s** function is multithread safe if no function in the current thread calls **setlocale** as long as this function is executing and the *mbstate* argument is not a null pointer.

Requirements

ROUTINE	REQUIRED HEADER
mbsrtowcs_s	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbrtowc
mbtowc, _mbtowc_l
mbstowcs_s, _mbstowcs_s_l
mbsinit

mbstowcs, _mbstowcs_l

3/1/2019 • 4 minutes to read • Edit Online

Converts a sequence of multibyte characters to a corresponding sequence of wide characters. More secure versions of these functions are available; see mbstowcs_s, _mbstowcs_s_l.

Syntax

```
size_t mbstowcs(
   wchar_t *wcstr,
   const char *mbstr,
   size_t count
);
size_t _mbstowcs_l(
   wchar_t *wcstr,
   const char *mbstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
size_t mbstowcs(
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count
); // C++ only
template <size_t size>
size_t _mbstowcs_l(
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

wcstr
The address of a sequence of wide characters.

mbstr
The address of a sequence of null terminated multibyte characters.

count
The maximum number of multibyte characters to convert.

locale
The locale to use.

Return Value

If **mbstowcs** successfully converts the source string, it returns the number of converted multibyte characters. If the *wcstr* argument is **NULL**, the function returns the required size (in wide characters) of the destination string. If **mbstowcs** encounters an invalid multibyte character, it returns -1. If the return value is *count*, the wide-character string is not null-terminated.

IMPORTANT

Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

Remarks

The **mbstowcs** function converts up to a maximum number of *count* multibyte characters pointed to by *mbstr* to a string of corresponding wide characters that are determined by the current locale. It stores the resulting wide-character string at the address represented by *wcstr*. The result is similar to a series of calls to mbtowc. If **mbstowcs** encounters the single-byte null character ('\0') either before or when *count* occurs, it converts the null character to a wide-character null character (L'\0') and stops. Thus the wide-character string at *wcstr* is null-terminated only if a null character is encountered during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior is undefined.

If the *wcstr* argument is **NULL**, **mbstowcs** returns the number of wide characters that would result from conversion, not including a null terminator. The source string must be null-terminated for the correct value to be returned. If you need the resulting wide character string to be null-terminated, add one to the returned value.

If the *mbstr* argument is **NULL**, or if *count* is > **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, errno is set to **EINVAL** and the function returns -1.

mbstowcs uses the current locale for any locale-dependent behavior; **_mbstowcs_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
mbstowcs	<stdlib.h>
_mbstowcs_l	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_mbstowcs.c
// compile with: /W3
// illustrates the behavior of the mbstowcs function

#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main( void )
{
    size_t size;
    int nChar = 2; // number of characters to convert
    int requiredSize;

    unsigned char    *pmbnull  = NULL;
    unsigned char    *pmbhello = NULL;
```

```
    unsigned char *pmbhello = NULL;
    char* localeInfo;

    wchar_t *pwchello = L"\x3042\x3043"; // 2 Hiragana characters
    wchar_t *pwc;

    /* Enable the Japanese locale and codepage */
    localeInfo = setlocale(LC_ALL, "Japanese_Japan.932");
    printf("Locale information set to %s\n", localeInfo);

    printf( "Convert to multibyte string:\n" );

    requiredSize = wcstombs( NULL, pwchello, 0); // C4996
    // Note: wcstombs is deprecated; consider using wcstombs_s
    printf("   Required Size: %d\n", requiredSize);

    /* Add one to leave room for the null terminator. */
    pmbhello = (unsigned char *)malloc( requiredSize + 1);
    if (! pmbhello)
    {
        printf("Memory allocation failure.\n");
        return 1;
    }
    size = wcstombs( pmbhello, pwchello, requiredSize + 1); // C4996
    // Note: wcstombs is deprecated; consider using wcstombs_s
    if (size == (size_t) (-1))
    {
        printf("Couldn't convert string. Code page 932 may"
                " not be available.\n");
        return 1;
    }
    printf( "   Number of bytes written to multibyte string: %u\n",
            (unsigned int) size );
    printf( "   Hex values of the" );
    printf( " multibyte characters: %#.2x %#.2x %#.2x %#.2x\n",
            pmbhello[0], pmbhello[1], pmbhello[2], pmbhello[3] );
    printf( "   Codepage 932 uses 0x81 to 0x9f as lead bytes.\n\n");

    printf( "Convert back to wide-character string:\n" );

    /* Assume we don't know the length of the multibyte string.
     Get the required size in characters, and allocate enough space. */

    requiredSize = mbstowcs(NULL, pmbhello, 0); // C4996
    /* Add one to leave room for the null terminator */
    pwc = (wchar_t *)malloc( (requiredSize + 1) * sizeof( wchar_t ));
    if (! pwc)
    {
        printf("Memory allocation failure.\n");
        return 1;
    }
    size = mbstowcs( pwc, pmbhello, requiredSize + 1); // C4996
    if (size == (size_t) (-1))
    {
       printf("Couldn't convert string--invalid multibyte character.\n");
    }
    printf( "   Characters converted: %u\n", (unsigned int)size );
    printf( "   Hex value of first 2" );
    printf( " wide characters: %#.4x %#.4x\n\n", pwc[0], pwc[1] );
    free(pwc);
    free(pmbhello);
}
```

```
Locale information set to Japanese_Japan.932
Convert to multibyte string:
   Required Size: 4
   Number of bytes written to multibyte string: 4
   Hex values of the  multibyte characters: 0x82 0xa0 0x82 0xa1
   Codepage 932 uses 0x81 to 0x9f as lead bytes.

Convert back to wide-character string:
   Characters converted: 2
   Hex value of first 2 wide characters: 0x3042 0x3043
```

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l
MultiByteToWideChar

mbstowcs_s, _mbstowcs_s_l

2/4/2019 • 2 minutes to read • Edit Online

Converts a sequence of multibyte characters to a corresponding sequence of wide characters. Versions of mbstowcs, _mbstowcs_l with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t mbstowcs_s(
   size_t *pReturnValue,
   wchar_t *wcstr,
   size_t sizeInWords,
   const char *mbstr,
   size_t count
);
errno_t _mbstowcs_s_l(
   size_t *pReturnValue,
   wchar_t *wcstr,
   size_t sizeInWords,
   const char *mbstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t mbstowcs_s(
   size_t *pReturnValue,
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbstowcs_s_l(
   size_t *pReturnValue,
   wchar_t (&wcstr)[size],
   const char *mbstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

pReturnValue
The number of characters converted.

wcstr
Address of buffer for the resulting converted wide character string.

sizeInWords
The size of the *wcstr* buffer in words.

mbstr
The address of a sequence of null terminated multibyte characters.

count
The maximum number of wide characters to store in the *wcstr* buffer, not including the terminating null, or _TRUNCATE.

locale

The locale to use.

Return Value

Zero if successful, an error code on failure.

ERROR CONDITION	RETURN VALUE AND ERRNO
wcstr is **NULL** and *sizeInWords* > 0	**EINVAL**
mbstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**
wcstr is not **NULL** and *sizeInWords* == 0	**EINVAL**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **mbstowcs_s** function converts a string of multibyte characters pointed to by *mbstr* into wide characters stored in the buffer pointed to by *wcstr*. The conversion will continue for each character until one of these conditions is met:

- A multibyte null character is encountered
- An invalid multibyte character is encountered
- The number of wide characters stored in the *wcstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **mbstowcs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **mbstowcs_s** successfully converts the source string, it puts the size in wide characters of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *wcstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *wcstr* is **NULL**, *count* is ignored, and *sizeInWords* must be 0.

If **mbstowcs_s** encounters an invalid multibyte character, it puts 0 in **pReturnValue*, sets the destination buffer to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *mbstr* and *wcstr* overlap, the behavior of **mbstowcs_s** is undefined.

> **IMPORTANT**
> Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of multibyte characters to convert.

mbstowcs_s uses the current locale for any locale-dependent behavior; **_mbstowcs_s_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length

automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
mbstowcs_s	<stdlib.h>
_mbstowcs_s_l	<stdlib.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
Locale
MultiByteToWideChar
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

mbtowc, _mbtowc_l

3/1/2019 • 2 minutes to read • Edit Online

Convert a multibyte character to a corresponding wide character.

Syntax

```
int mbtowc(
   wchar_t *wchar,
   const char *mbchar,
   size_t count
);
int _mbtowc_l(
   wchar_t *wchar,
   const char *mbchar,
   size_t count,
   _locale_t locale
);
```

Parameters

wchar
Address of a wide character (type **wchar_t**).

mbchar
Address of a sequence of bytes (a multibyte character).

count
Number of bytes to check.

locale
The locale to use.

Return Value

If **mbchar** is not **NULL** and if the object that *mbchar* points to forms a valid multibyte character, **mbtowc** returns the length in bytes of the multibyte character. If *mbchar* is **NULL** or the object that it points to is a wide-character null character (L'\0'), the function returns 0. If the object that *mbchar* points to does not form a valid multibyte character within the first *count* characters, it returns -1.

Remarks

The **mbtowc** function converts *count* or fewer bytes pointed to by *mbchar*, if *mbchar* is not **NULL**, to a corresponding wide character. **mbtowc** stores the resulting wide character at *wchar,* if *wchar* is not **NULL**. **mbtowc** does not examine more than **MB_CUR_MAX** bytes. **mbtowc** uses the current locale for locale-dependent behavior; **_mbtowc_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
mbtowc	<stdlib.h>
_mbtowc_l	<stdlib.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_mbtowc.c
// Illustrates the behavior of the mbtowc function

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int      i;
    char    *pmbc    = (char *)malloc( sizeof( char ) );
    wchar_t  wc      = L'a';
    wchar_t *pwcnull = NULL;
    wchar_t *pwc     = (wchar_t *)malloc( sizeof( wchar_t ) );
    printf( "Convert a wide character to multibyte character:\n" );
    wctomb_s( &i, pmbc, sizeof(char), wc );
    printf( "  Characters converted: %u\n", i );
    printf( "  Multibyte character: %x\n\n", *pmbc );

    printf( "Convert multibyte character back to a wide "
            "character:\n" );
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "   Bytes converted: %u\n", i );
    printf( "   Wide character: %x\n\n", *pwc );
    printf( "Attempt to convert when target is NULL\n" );
    printf( "   returns the length of the multibyte character:\n" );
    i = mbtowc( pwcnull, pmbc, MB_CUR_MAX );
    printf( "   Length of multibyte character: %u\n\n", i );

    printf( "Attempt to convert a NULL pointer to a" );
    printf( " wide character:\n" );
    pmbc = NULL;
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "   Bytes converted: %u\n", i );
}
```

```
Convert a wide character to multibyte character:
   Characters converted: 1
   Multibyte character: 61

Convert multibyte character back to a wide character:
   Bytes converted: 1
   Wide character: 61

Attempt to convert when target is NULL
   returns the length of the multibyte character:
   Length of multibyte character: 1

Attempt to convert a NULL pointer to a wide character:
   Bytes converted: 0
```

See also

Data Conversion
MultiByteToWideChar
Locale
Interpretation of Multibyte-Character Sequences
_mbclen, mblen, _mblen_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

memccpy

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _memccpy instead.

_memccpy

10/31/2018 • 2 minutes to read • Edit Online

Copies characters from a buffer.

Syntax

```
void *_memccpy(
   void *dest,
   const void *src,
   int c,
   size_t count
);
```

Parameters

dest
Pointer to the destination.

src
Pointer to the source.

c
Last character to copy.

count
Number of characters.

Return Value

If the character *c* is copied, **_memccpy** returns a pointer to the char in *dest* that immediately follows the character. If *c* is not copied, it returns **NULL**.

Remarks

The **_memccpy** function copies 0 or more characters of *src* to *dest*, halting when the character *c* has been copied or when *count* characters have been copied, whichever comes first.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

Requirements

ROUTINE	REQUIRED HEADER
_memccpy	<memory.h> or <string.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memccpy.c

#include <memory.h>
#include <stdio.h>
#include <string.h>

char string1[60] = "The quick brown dog jumps over the lazy fox";

int main( void )
{
   char buffer[61];
   char *pdest;

   printf( "Function: _memccpy 60 characters or to character 's'\n" );
   printf( "Source: %s\n", string1 );
   pdest = _memccpy( buffer, string1, 's', 60 );
   *pdest = '\0';
   printf( "Result: %s\n", buffer );
   printf( "Length: %d characters\n", strlen( buffer ) );
}
```

Output

```
Function: _memccpy 60 characters or to character 's'
Source: The quick brown dog jumps over the lazy fox
Result: The quick brown dog jumps
Length: 25 characters
```

See also

Buffer Manipulation
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset

memchr, wmemchr

4/2/2019 • 2 minutes to read • Edit Online

Find characters in a buffer.

Syntax

```
void *memchr(
   const void *buffer,
   int c,
   size_t count
); // C only
void *memchr(
   void *buffer,
   int c,
   size_t count
); // C++ only
const void *memchr(
   const void *buffer,
   int c,
   size_t count
); // C++ only
wchar_t *wmemchr(
   const wchar_t * buffer,
   wchar_t c,
   size_t count
); // C only
wchar_t *wmemchr(
   wchar_t * buffer,
   wchar_t c,
   size_t count
); // C++ only
const wchar_t *wmemchr(
   const wchar_t * buffer,
   wchar_t c,
   size_t count
); // C++ only
```

Parameters

buffer

Pointer to buffer.

c

Character to look for.

count

Number of characters to check.

Return Value

If successful, returns a pointer to the first location of *c* in *buffer*. Otherwise it returns NULL.

Remarks

`memchr` and `wmemchr` look for the first occurrence of *c* in the first *count* characters of *buffer*. It stops when it finds *c* or when it has checked the first *count* characters.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CRT_CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads in C++, define the symbol _CONST_RETURN.

Requirements

ROUTINE	REQUIRED HEADER
`memchr`	<memory.h> or <string.h>
`wmemchr`	<wchar.h>

For more information about compatibility, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memchr.c

#include <memory.h>
#include <stdio.h>

int  ch = 'r';
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int result;
   printf( "String to be searched:\n             %s\n", string );
   printf( "             %s\n             %s\n\n", fmt1, fmt2 );

   printf( "Search char: %c\n", ch );
   pdest = memchr( string, ch, sizeof( string ) );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf( "Result:      %c found at position %d\n", ch, result );
   else
      printf( "Result:      %c not found\n" );
}
```

Output

```
String to be searched:
             The quick brown dog jumps over the lazy fox
                      1         2         3         4         5
             12345678901234567890123456789012345678901234567890

Search char: r
Result:      r found at position 12
```

See also

Buffer Manipulation
_memccpy
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset
strchr, wcschr, _mbschr, _mbschr_l

memcmp, wmemcmp

3/1/2019 • 2 minutes to read • Edit Online

Compares characters in two buffers.

Syntax

```
int memcmp(
   const void *buffer1,
   const void *buffer2,
   size_t count
);
int wmemcmp(
   const wchar_t * buffer1,
   const wchar_t * buffer2,
   size_t count
);
```

Parameters

buffer1
First buffer.

buffer2
Second buffer.

count
Number of characters to compare. (Compares bytes for **memcmp**, wide characters for **wmemcmp**).

Return Value

The return value indicates the relationship between the buffers.

RETURN VALUE	RELATIONSHIP OF FIRST *COUNT* CHARACTERS OF BUF1 AND BUF2
< 0	*buffer1* less than *buffer2*
0	*buffer1* identical to *buffer2*
> 0	*buffer1* greater than *buffer2*

Remarks

Compares the first *count* characters of *buffer1* and *buffer2* and returns a value that indicates their relationship. The sign of a non-zero return value is the sign of the difference between the first differing pair of values in the buffers. The values are interpreted as **unsigned char** for **memcmp**, and as **wchar_t** for **wmemcmp**.

Requirements

ROUTINE	REQUIRED HEADER
memcmp	<memory.h> or <string.h>
wmemcmp	<wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time library.

Example

```
// crt_memcmp.c
/* This program uses memcmp to compare
* the strings named first and second. If the first
* 19 bytes of the strings are equal, the program
* considers the strings to be equal.
*/

#include <string.h>
#include <stdio.h>

int main( void )
{
   char first[]  = "12345678901234567890";
   char second[] = "12345678901234567891";
   int int_arr1[] = {1,2,3,4};
   int int_arr2[] = {1,2,3,4};
   int result;

   printf( "Compare '%.19s' to '%.19s':\n", first, second );
   result = memcmp( first, second, 19 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else
      printf( "First is greater than second.\n" );

   printf( "Compare '%d,%d' to '%d,%d':\n", int_arr1[0], int_arr1[1], int_arr2[0], int_arr2[1]);
   result = memcmp( int_arr1, int_arr2, sizeof(int) * 2 );
   if( result < 0 )
      printf( "int_arr1 is less than int_arr2.\n" );
   else if( result == 0 )
      printf( "int_arr1 is equal to int_arr2.\n" );
   else
      printf( "int_arr1 is greater than int_arr2.\n" );
}
```

```
Compare '1234567890123456789' to '1234567890123456789':
First is equal to second.
Compare '1,2' to '1,2':
int_arr1 is equal to int_arr2.
```

See also

Buffer Manipulation

_memccpy
memchr, wmemchr
memcpy, wmemcpy
memset, wmemset
strcmp, wcscmp, _mbscmp
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

memcpy, wmemcpy

3/1/2019 • 2 minutes to read • Edit Online

Copies bytes between buffers. More secure versions of these functions are available; see memcpy_s, wmemcpy_s.

Syntax

```
void *memcpy(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemcpy(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest
New buffer.

src
Buffer to copy from.

count
Number of characters to copy.

Return Value

The value of *dest*.

Remarks

memcpy copies *count* bytes from *src* to *dest*; **wmemcpy** copies *count* wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy** is undefined. Use **memmove** to handle overlapping regions.

> **IMPORTANT**
> Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

IMPORTANT

Because so many buffer overruns, and thus potential security exploits, have been traced to improper usage of **memcpy**, this function is listed among the "banned" functions by the Security Development Lifecycle (SDL). You may observe that some VC++ library classes continue to use **memcpy**. Furthermore, you may observe that the VC++ compiler optimizer sometimes emits calls to **memcpy**. The Visual C++ product is developed in accordance with the SDL process, and thus usage of this banned function has been closely evaluated. In the case of library use of it, the calls have been carefully scrutinized to ensure that buffer overruns will not be allowed through these calls. In the case of the compiler, sometimes certain code patterns are recognized as identical to the pattern of **memcpy**, and are thus replaced with a call to the function. In such cases, the use of **memcpy** is no more unsafe than the original instructions would have been; they have simply been optimized to a call to the performance-tuned **memcpy** function. Just as the use of "safe" CRT functions doesn't guarantee safety (they just make it harder to be unsafe), the use of "banned" functions doesn't guarantee danger (they just require greater scrutiny to ensure safety).

Because **memcpy** usage by the VC++ compiler and libraries has been so carefully scrutinized, these calls are permitted within code that is otherwise compliant with SDL. **memcpy** calls introduced in application source code are only compliant with the SDL when that use has been reviewed by security experts.

The **memcpy** and **wmemcpy** functions will only be deprecated if the constant **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion statement in order for the functions to be deprecated, such as in the example below:

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <memory.h>
```

or

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <wchar.h>
```

Requirements

ROUTINE	REQUIRED HEADER
memcpy	<memory.h> or <string.h>
wmemcpy	<wchar.h>

For additional compatibility information, see Compatibility.

Example

See memmove for a sample of how to use **memcpy**.

See also

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memmove, wmemmove
memset, wmemset

strcpy_s, wcscpy_s, _mbscpy_s

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

memcpy_s, wmemcpy_s

3/1/2019 • 2 minutes to read • Edit Online

Copies bytes between buffers. These are versions of memcpy, wmemcpy with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t memcpy_s(
   void *dest,
   size_t destSize,
   const void *src,
   size_t count
);
errno_t wmemcpy_s(
   wchar_t *dest,
   size_t destSize,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest
New buffer.

destSize
Size of the destination buffer, in bytes for memcpy_s and wide characters (wchar_t) for wmemcpy_s.

src
Buffer to copy from.

count
Number of characters to copy.

Return Value

Zero if successful; an error code on failure.

Error Conditions

DEST	*DESTSIZE*	*SRC*	*COUNT*	RETURN VALUE	CONTENTS OF *DEST*
any	any	any	0	0	Not modified
NULL	any	any	non-zero	**EINVAL**	Not modified
any	any	**NULL**	non-zero	**EINVAL**	*dest* is zeroed out
any	< *count*	any	non-zero	**ERANGE**	*dest* is zeroed out

Remarks

memcpy_s copies *count* bytes from *src* to *dest*; **wmemcpy_s** copies *count* wide characters (two bytes). If the source and destination overlap, the behavior of **memcpy_s** is undefined. Use **memmove_s** to handle overlapping regions.

These functions validate their parameters. If *count* is non-zero and *dest* or *src* is a null pointer, or *destSize* is smaller than *count*, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** or **ERANGE** and set **errno** to the return value.

Requirements

ROUTINE	REQUIRED HEADER
memcpy_s	<memory.h> or <string.h>
wmemcpy_s	<wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_memcpy_s.c
// Copy memory in a more secure way.

#include <memory.h>
#include <stdio.h>

int main()
{
   int a1[10], a2[100], i;
   errno_t err;

   // Populate a2 with squares of integers
   for (i = 0; i < 100; i++)
   {
      a2[i] = i*i;
   }

   // Tell memcpy_s to copy 10 ints (40 bytes), giving
   // the size of the a1 array (also 40 bytes).
   err = memcpy_s(a1, sizeof(a1), a2, 10 * sizeof (int) );
   if (err)
   {
      printf("Error executing memcpy_s.\n");
   }
   else
   {
     for (i = 0; i < 10; i++)
       printf("%d ", a1[i]);
   }
   printf("\n");
}
```

```
0 1 4 9 16 25 36 49 64 81
```

See also

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memmove, wmemmove
memset, wmemset
strcpy, wcscpy, _mbscpy
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

memicmp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _memicmp instead.

_memicmp, _memicmp_l

10/31/2018 • 2 minutes to read • Edit Online

Compares characters in two buffers (case-insensitive).

Syntax

```
int _memicmp(
   const void *buffer1,
   const void *buffer2,
   size_t count
);
int _memicmp_l(
   const void *buffer1,
   const void *buffer2,
   size_t count,
   _locale_t locale
);
```

Parameters

buffer1
First buffer.

buffer2
Second buffer.

count
Number of characters.

locale
Locale to use.

Return Value

The return value indicates the relationship between the buffers.

RETURN VALUE	RELATIONSHIP OF FIRST COUNT BYTES OF BUF1 AND BUF2
< 0	*buffer1* less than *buffer2*.
0	*buffer1* identical to *buffer2*.
> 0	*buffer1* greater than *buffer2*.
_NLSCMPERROR	An error occurred.

Remarks

The **_memicmp** function compares the first *count* characters of the two buffers *buffer1* and *buffer2* byte by byte. The comparison is not case-sensitive.

If either *buffer1* or *buffer2* is a null pointer, this function invokes an invalid parameter handler, as described in

Parameter Validation. If execution is allowed to continue, the function returns **_NLSCMPERROR** and sets **errno** to **EINVAL**.

_memicmp uses the current locale for locale-dependent behavior; **_memicmp_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_memicmp	<memory.h> or <string.h>
_memicmp_l	<memory.h> or <string.h>

For more compatibility information, see Compatibility.

Example

```
// crt_memicmp.c
// This program uses _memicmp to compare
// the first 29 letters of the strings named first and
// second without regard to the case of the letters.

#include <memory.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
   int result;
   char first[] = "Those Who Will Not Learn from History";
   char second[] = "THOSE WHO WILL NOT LEARN FROM their mistakes";
   // Note that the 29th character is right here ^

   printf( "Compare '%.29s' to '%.29s'\n", first, second );
   result = _memicmp( first, second, 29 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

```
Compare 'Those Who Will Not Learn from' to 'THOSE WHO WILL NOT LEARN FROM'
First is equal to second.
```

See also

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
memset, wmemset
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

memmove, wmemmove

3/1/2019 • 2 minutes to read • Edit Online

Moves one buffer to another. More secure versions of these functions are available; see memmove_s, wmemmove_s.

Syntax

```
void *memmove(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemmove(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest
Destination object.

src
Source object.

count
Number of bytes (**memmove**) or characters (**wmemmove**) to copy.

Return Value

The value of *dest*.

Remarks

Copies *count* bytes (**memmove**) or characters (**wmemmove**) from *src* to *dest*. If some regions of the source area and the destination overlap, both functions ensure that the original source bytes in the overlapping region are copied before being overwritten.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

The **memmove** and **wmemmove** functions will only be deprecated if the constant **_CRT_SECURE_DEPRECATE_MEMORY** is defined prior to the inclusion statement in order for the functions to be deprecated, such as in the example below:

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <string.h>
```

or

```
#define _CRT_SECURE_DEPRECATE_MEMORY
#include <wchar.h>
```

Requirements

ROUTINE	REQUIRED HEADER
memmove	<string.h>
wmemmove	<wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_memcpy.c
// Illustrate overlapping copy: memmove
// always handles it correctly; memcpy may handle
// it correctly.
//

#include <memory.h>
#include <string.h>
#include <stdio.h>

char str1[7] = "aabbcc";

int main( void )
{
   printf( "The string: %s\n", str1 );
   memcpy( str1 + 2, str1, 4 );
   printf( "New string: %s\n", str1 );

   strcpy_s( str1, sizeof(str1), "aabbcc" );   // reset string

   printf( "The string: %s\n", str1 );
   memmove( str1 + 2, str1, 4 );
   printf( "New string: %s\n", str1 );
}
```

```
The string: aabbcc
New string: aaaabb
The string: aabbcc
New string: aaaabb
```

See also

Buffer Manipulation
_memccpy
memcpy, wmemcpy
strcpy, wcscpy, _mbscpy
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

memmove_s, wmemmove_s

3/1/2019 • 2 minutes to read • Edit Online

Moves one buffer to another. These are versions of memmove, wmemmove with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t memmove_s(
   void *dest,
   size_t numberOfElements,
   const void *src,
   size_t count
);
errno_t wmemmove_s(
   wchar_t *dest,
   size_t numberOfElements,
   const wchar_t *src,
   size_t count
);
```

Parameters

dest
Destination object.

numberOfElements
Size of the destination buffer.

src
Source object.

count
Number of bytes (**memmove_s**) or characters (**wmemmove_s**) to copy.

Return Value

Zero if successful; an error code on failure

Error Conditions

DEST	*NUMBEROFELEMENTS*	*SRC*	RETURN VALUE	CONTENTS OF *DEST*
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	not modified
any	< *count*	any	**ERANGE**	not modified

Remarks

Copies *count* bytes of characters from *src* to *dest*. If some regions of the source area and the destination overlap, **memmove_s** ensures that the original source bytes in the overlapping region are copied before being

overwritten.

If *dest* or if *src* is a null pointer, or if the destination string is too small, these functions invoke an invalid parameter handler, as described in Parameter Validation . If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
memmove_s	<string.h>
wmemmove_s	<wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_memmove_s.c
//
// The program demonstrates the
// memmove_s function which works as expected
// for moving overlapping regions.

#include <stdio.h>
#include <string.h>

int main()
{
   char str[] = "0123456789";

   printf("Before: %s\n", str);

   // Move six bytes from the start of the string
   // to a new position shifted by one byte. To protect against
   // buffer overrun, the secure version of memmove requires the
   // the length of the destination string to be specified.

   memmove_s((str + 1), strnlen(str + 1, 10), str, 6);

   printf_s(" After: %s\n", str);
}
```

Output

```
Before: 0123456789
After: 0012345789
```

See also

Buffer Manipulation
_memccpy
memcpy, wmemcpy
strcpy_s, wcscpy_s, _mbscpy_s
strcpy, wcscpy, _mbscpy
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

memset, wmemset

3/1/2019 • 2 minutes to read • Edit Online

Sets buffers to a specified character.

Syntax

```
void *memset(
   void *dest,
   int c,
   size_t count
);
wchar_t *wmemset(
   wchar_t *dest,
   wchar_t c,
   size_t count
);
```

Parameters

dest
Pointer to destination.

c
Character to set.

count
Number of characters.

Return Value

The value of *dest*.

Remarks

Sets the first *count* characters of *dest* to the character *c*.

Security Note Make sure that the destination buffer has enough room for at least *count* characters. For more information, see Avoiding Buffer Overruns.

Requirements

ROUTINE	REQUIRED HEADER
memset	<memory.h> or <string.h>
wmemset	<wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_memset.c
/* This program uses memset to
* set the first four chars of buffer to "*".
*/

#include <memory.h>
#include <stdio.h>

int main( void )
{
   char buffer[] = "This is a test of the memset function";

   printf( "Before: %s\n", buffer );
   memset( buffer, '*', 4 );
   printf( "After:  %s\n", buffer );
}
```

Output

```
Before: This is a test of the memset function
After:  **** is a test of the memset function
```

Here's an example of the use of wmemset:

```
// crt_wmemset.c
/* This program uses memset to
* set the first four chars of buffer to "*".
*/

#include <wchar.h>
#include <stdio.h>

int main( void )
{
   wchar_t buffer[] = L"This is a test of the wmemset function";

   wprintf( L"Before: %s\n", buffer );
   wmemset( buffer, '*', 4 );
   wprintf( L"After:  %s\n", buffer );
}
```

Output

```
Before: This is a test of the wmemset function
After:  **** is a test of the wmemset function
```

See also

Buffer Manipulation
_memccpy
memchr, wmemchr
memcmp, wmemcmp
memcpy, wmemcpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

__min

10/31/2018 • 2 minutes to read • Edit Online

A preprocessor macro that returns the smaller of two values.

Syntax

```
#define __min(a,b) (((a) < (b)) ? (a) : (b))
```

Parameters

a, *b*
Values of any type that the **<** operator works on.

Return Value

The smaller of the two arguments.

Remarks

The **__min** macro compares two values and returns the value of the smaller one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

The argument returned is evaluated twice by the macro. This can lead to unexpected results if the argument is an expression that alters its value when it is evaluated, such as `*p++`.

Requirements

ROUTINE	REQUIRED HEADER
__min	<stdlib.h>

Example

```
// crt_minmax.c

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   int a = 10;
   int b = 21;

   printf( "The larger of %d and %d is %d\n",  a, b, __max( a, b ) );
   printf( "The smaller of %d and %d is %d\n", a, b, __min( a, b ) );
}
```

```
The larger of 10 and 21 is 21
The smaller of 10 and 21 is 10
```

See also

Floating-Point Support
__max

mkdir

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _mkdir instead.

_mkdir, _wmkdir

10/31/2018 • 2 minutes to read • Edit Online

Creates a new directory.

Syntax

```
int _mkdir(
   const char *dirname
);
int _wmkdir(
   const wchar_t *dirname
);
```

Parameters

dirname
Path for a new directory.

Return Value

Each of these functions returns the value 0 if the new directory was created. On an error, the function returns -1 and sets **errno** as follows.

EEXIST Directory was not created because *dirname* is the name of an existing file, directory, or device.

ENOENT Path was not found.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_mkdir** function creates a new directory with the specified *dirname*. **_mkdir** can create only one new directory per call, so only the last component of *dirname* can name a new directory. **_mkdir** does not translate path delimiters. In Windows NT, both the backslash (\) and the forward slash (/) are valid path delimiters in character strings in run-time routines.

_wmkdir is a wide-character version of **_mkdir**; the *dirname* argument to **_wmkdir** is a wide-character string. **_wmkdir** and **_mkdir** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tmkdir	**_mkdir**	**_mkdir**	**_wmkdir**

Requirements

ROUTINE	REQUIRED HEADER
_mkdir	<direct.h>
_wmkdir	<direct.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_makedir.c

#include <direct.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   if( _mkdir( "\\testtmp" ) == 0 )
   {
      printf( "Directory '\\testtmp' was successfully created\n" );
      system( "dir \\testtmp" );
      if( _rmdir( "\\testtmp" ) == 0 )
        printf( "Directory '\\testtmp' was successfully removed\n"  );
      else
         printf( "Problem removing directory '\\testtmp'\n" );
   }
   else
      printf( "Problem creating directory '\\testtmp'\n" );
}
```

Sample Output

```
Directory '\testtmp' was successfully created
Volume in drive C has no label.
Volume Serial Number is E078-087A

Directory of C:\testtmp

02/12/2002  09:56a      <DIR>          .
02/12/2002  09:56a      <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  15,498,690,560 bytes free
Directory '\testtmp' was successfully removed
```

See also

Directory Control
_chdir, _wchdir
_rmdir, _wrmdir

_mkgmtime, _mkgmtime32, _mkgmtime64

10/31/2018 • 3 minutes to read • Edit Online

Converts a UTC time represented by a **struct tm** to a UTC time represented by a **time_t** type.

Syntax

```
time_t _mkgmtime(
   struct tm* timeptr
);
__time32_t _mkgmtime32(
   struct tm* timeptr
);
__time64_t _mkgmtime64(
   struct tm* timeptr
);
```

Parameters

timeptr
A pointer to the UTC time as a **struct tm** to convert.

Return Value

A quantity of type **__time32_t** or **__time64_t** representing the number of seconds elapsed since midnight, January 1, 1970, in Coordinated Universal Time (UTC). If the date is out of range (see the Remarks section) or the input cannot be interpreted as a valid time, the return value is -1.

Remarks

The **_mkgmtime32** and **_mkgmtime64** functions convert a UTC time to a **__time32_t** or **__time64_t** type representing the time in UTC. To convert a local time to UTC time, use **mktime**, **_mktime32**, and **_mktime64** instead.

_mkgmtime is an inline function that evaluates to **_mkgmtime64**, and **time_t** is equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. This is not recommended because your application might fail after January 18, 2038 (the maximum range of a 32-bit **time_t**), and it is not allowed at all on 64-bit platforms.

The time structure passed in will be changed as follows, in the same way as they are changed with the **_mktime** functions: the **tm_wday** and **tm_yday** fields are set to new values based on the values of **tm_mday** and **tm_year**. When specifying a **tm** structure time, set the **tm_isdst** field to:

- Zero (0) to indicate that standard time is in effect.
- A value greater than 0 to indicate that daylight saving time is in effect.
- A value less than zero to have the C run-time library code compute whether standard time or daylight saving time is in effect.

The C run-time library uses the TZ environment variable to determine the correct daylight savings time. If TZ is not set, the operating system is queried to get the correct regional daylight savings time behavior. **tm_isdst** is a required field. If not set, its value is undefined and the return value from **mktime** is unpredictable.

The range of the **_mkgmtime32** function is from midnight, January 1, 1970, UTC to 23:59:59 January 18, 2038, UTC. The range of **_mkgmtime64** is from midnight, January 1, 1970, UTC to 23:59:59, December 31, 3000, UTC. An out-of-range date results in a return value of -1. The range of **_mkgmtime** depends on whether **_USE_32BIT_TIME_T** is defined. If not defined (the default) the range is that of **_mkgmtime64**; otherwise, the range is limited to the 32-bit range of **_mkgmtime32**.

Note that **gmtime** and **localtime** use a single statically allocated buffer for the conversion. If you supply this buffer to **mkgmtime**, the previous contents are destroyed.

Example

```
// crt_mkgmtime.c
#include <stdio.h>
#include <time.h>

int main()
{
    struct tm t1, t2;
    time_t now, mytime, gmtime;
    char buff[30];

    time( & now );

    _localtime64_s( &t1, &now );
    _gmtime64_s( &t2, &now );

    mytime = mktime(&t1);
    gmtime = _mkgmtime(&t2);

    printf("Seconds since midnight, January 1, 1970\n");
    printf("My time: %I64d\nGM time (UTC): %I64d\n\n", mytime, gmtime);

    /* Use asctime_s to display these times. */

    _localtime64_s( &t1, &mytime );
    asctime_s( buff, sizeof(buff), &t1 );
    printf( "Local Time: %s\n", buff );

    _gmtime64_s( &t2, &gmtime );
    asctime_s( buff, sizeof(buff), &t2 );
    printf( "Greenwich Mean Time: %s\n", buff );

}
```

Sample Output

```
Seconds since midnight, January 1, 1970
My time: 1171588492
GM time (UTC): 1171588492

Local Time: Thu Feb 15 17:14:52 2007

Greenwich Mean Time: Fri Feb 16 01:14:52 2007
```

The following example shows how the incomplete structure is filled out with the computed values of the day of the week and the day of the year.

```
// crt_mkgmtime2.c
#include <stdio.h>
#include <time.h>
#include <memory.h>

int main()
{
    struct tm t1, t2;
    time_t gmtime;
    char buff[30];

    memset(&t1, 0, sizeof(struct tm));
    memset(&t2, 0, sizeof(struct tm));

    t1.tm_mon = 1;
    t1.tm_isdst = 0;
    t1.tm_year = 103;
    t1.tm_mday = 12;

    // The day of the week and year will be incorrect in the output here.
    asctime_s( buff, sizeof(buff), &t1);
    printf("Before calling _mkgmtime, t1 = %s t.tm_yday = %d\n",
            buff, t1.tm_yday );

    gmtime = _mkgmtime(&t1);

    // The correct day of the week and year were determined.
    asctime_s( buff, sizeof(buff), &t1);
    printf("After calling _mkgmtime, t1 = %s t.tm_yday = %d\n",
            buff, t1.tm_yday );

}
```

Output

```
Before calling _mkgmtime, t1 = Sun Feb 12 00:00:00 2003
t.tm_yday = 0
After calling _mkgmtime, t1 = Wed Feb 12 00:00:00 2003
t.tm_yday = 42
```

See also

Time Management
asctime, _wasctime
asctime_s, _wasctime_s
gmtime, _gmtime32, _gmtime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
mktime, _mktime32, _mktime64
time, _time32, _time64

mktemp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _mktemp or security-enhanced _mktemp_s instead.

_mktemp, _wmktemp

11/8/2018 • 4 minutes to read • Edit Online

Creates a unique file name. More secure versions of these functions are available; see _mktemp_s, _wmktemp_s.

Syntax

```
char *_mktemp(
   char *nameTemplate
);
wchar_t *_wmktemp(
   wchar_t *nameTemplate
);
template <size_t size>
char *_mktemp(
   char (&nameTemplate)[size]
); // C++ only
template <size_t size>
wchar_t *_wmktemp(
   wchar_t (&nameTemplate)[size]
); // C++ only
```

Parameters

nameTemplate
File name pattern.

Return Value

Each of these functions returns a pointer to the modified nameTemplate. The function returns **NULL** if *nameTemplate* is badly formed or no more unique names can be created from the given nameTemplate.

Remarks

The **_mktemp** function creates a unique file name by modifying the *nameTemplate* argument. **_mktemp** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use by the run-time system. **_wmktemp** is a wide-character version of **_mktemp**; the argument and return value of **_wmktemp** are wide-character strings. **_wmktemp** and **_mktemp** behave identically otherwise, except that **_wmktemp** does not handle multibyte-character strings.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tmktemp	**_mktemp**	**_mktemp**	**_wmktemp**

The *nameTemplate* argument has the form *base*XXXXXX, where *base* is the part of the new file name that you supply and each X is a placeholder for a character supplied by **_mktemp**. Each placeholder character in *nameTemplate* must be an uppercase X. **_mktemp** preserves *base* and replaces the first trailing X with an alphabetic character. **_mktemp** replaces the following trailing X's with a five-digit value; this value is a unique number identifying the calling process, or in multithreaded programs, the calling thread.

Each successful call to **_mktemp** modifies *nameTemplate*. In each subsequent call from the same process or thread with the same *nameTemplate* argument, **_mktemp** checks for file names that match names returned by **_mktemp** in previous calls. If no file exists for a given name, **_mktemp** returns that name. If files exist for all previously returned names, **_mktemp** creates a new name by replacing the alphabetic character it used in the previously returned name with the next available lowercase letter, in order, from 'a' through 'z'. For example, if *base* is:

```
fn
```

and the five-digit value supplied by **_mktemp** is 12345, the first name returned is:

```
fna12345
```

If this name is used to create file FNA12345 and this file still exists, the next name returned on a call from the same process or thread with the same *base* for *nameTemplate* is:

```
fnb12345
```

If FNA12345 does not exist, the next name returned is again:

```
fna12345
```

_mktemp can create a maximum of 26 unique file names for any given combination of *base* and *nameTemplate* values. Therefore, FNZ12345 is the last unique file name **_mktemp** can create for the *base* and *nameTemplate* values used in this example.

On failure, **errno** is set. If *nameTemplate* has an invalid format (for example, fewer than 6 X's), **errno** is set to **EINVAL**. If **_mktemp** is unable to create a unique name because all 26 possible file names already exist, **_mktemp** sets nameTemplate to an empty string and returns **EEXIST**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
_mktemp	<io.h>
_wmktemp	<io.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mktemp.c
// compile with: /W3
/* The program uses _mktemp to create
* unique filenames. It opens each filename
* to ensure that the next name is unique.
*/

#include <io.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

char *template = "fnXXXXXX";
char *result;
char names[27][9];

int main( void )
{
   int i;
   FILE *fp;

   for( i = 0; i < 27; i++ )
   {
      strcpy_s( names[i], sizeof( names[i] ), template );
      /* Attempt to find a unique filename: */
      result = _mktemp( names[i] );  // C4996
      // Note: _mktemp is deprecated; consider using _mktemp_s instead
      if( result == NULL )
      {
         printf( "Problem creating the template\n" );
         if (errno == EINVAL)
         {
             printf( "Bad parameter\n");
         }
         else if (errno == EEXIST)
         {
             printf( "Out of unique filenames\n");
         }
      }
      else
      {
         fopen_s( &fp, result, "w" );
         if( fp != NULL )
            printf( "Unique filename is %s\n", result );
         else
            printf( "Cannot open %s\n", result );
         fclose( fp );
      }
   }
}
```

```
Unique filename is fna03912
Unique filename is fnb03912
Unique filename is fnc03912
Unique filename is fnd03912
Unique filename is fne03912
Unique filename is fnf03912
Unique filename is fng03912
Unique filename is fnh03912
Unique filename is fni03912
Unique filename is fnj03912
Unique filename is fnk03912
Unique filename is fnl03912
Unique filename is fnm03912
Unique filename is fnn03912
Unique filename is fno03912
Unique filename is fnp03912
Unique filename is fnq03912
Unique filename is fnr03912
Unique filename is fns03912
Unique filename is fnt03912
Unique filename is fnu03912
Unique filename is fnv03912
Unique filename is fnw03912
Unique filename is fnx03912
Unique filename is fny03912
Unique filename is fnz03912
Problem creating the template.
Out of unique filenames.
```

See also

File Handling
fopen, _wfopen
_getmbcp
_getpid
_open, _wopen
_setmbcp
_tempnam, _wtempnam, tmpnam, _wtmpnam
tmpfile

_mktemp_s, _wmktemp_s

10/31/2018 • 3 minutes to read • Edit Online

Creates a unique file name. These are versions of _mktemp, _wmktemp with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _mktemp_s(
   char *nameTemplate,
   size_t sizeInChars
);
errno_t _wmktemp_s(
   wchar_t *nameTemplate,
   size_t sizeInChars
);
template <size_t size>
errno_t _mktemp_s(
   char (&nameTemplate)[size]
); // C++ only
template <size_t size>
errno_t _wmktemp_s(
   wchar_t (&nameTemplate)[size]
); // C++ only
```

Parameters

nameTemplate
File name pattern.

sizeInChars
Size of the buffer in single-byte characters in **_mktemp_s**; wide characters in **_wmktemp_s**, including the null terminator.

Return Value

Both of these functions return zero on success; an error code on failure.

Error Conditions

NAMETEMPLATE	*SIZEINCHARS*	RETURN VALUE	NEW VALUE IN *NAMETEMPLATE*
NULL	any	**EINVAL**	**NULL**
Incorrect format (see Remarks section for correct format)	any	**EINVAL**	empty string
any	<= number of X's	**EINVAL**	empty string

If any of the above error conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the functions returns **EINVAL**.

Remarks

The **_mktemp_s** function creates a unique file name by modifying the *nameTemplate* argument, so that after the call, the *nameTemplate* pointer points to a string containing the new file name. **_mktemp_s** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use by the run-time system. **_wmktemp_s** is a wide-character version of **_mktemp_s**; the argument of **_wmktemp_s** is a wide-character string. **_wmktemp_s** and **_mktemp_s** behave identically otherwise, except that **_wmktemp_s** does not handle multibyte-character strings.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tmktemp_s	**_mktemp_s**	**_mktemp_s**	**_wmktemp_s**

The *nameTemplate* argument has the form **baseXXXXXX**, where *base* is the part of the new file name that you supply and each X is a placeholder for a character supplied by **_mktemp_s**. Each placeholder character in *nameTemplate* must be an uppercase X. **_mktemp_s** preserves *base* and replaces the first trailing X with an alphabetic character. **_mktemp_s** replaces the following trailing X's with a five-digit value; this value is a unique number identifying the calling process, or in multithreaded programs, the calling thread.

Each successful call to **_mktemp_s** modifies *nameTemplate*. In each subsequent call from the same process or thread with the same *nameTemplate* argument, **_mktemp_s** checks for file names that match names returned by **_mktemp_s** in previous calls. If no file exists for a given name, **_mktemp_s** returns that name. If files exist for all previously returned names, **_mktemp_s** creates a new name by replacing the alphabetic character it used in the previously returned name with the next available lowercase letter, in order, from 'a' through 'z'. For example, if *base* is:

```
fn
```

and the five-digit value supplied by **_mktemp_s** is 12345, the first name returned is:

```
fna12345
```

If this name is used to create file FNA12345 and this file still exists, the next name returned on a call from the same process or thread with the same *base* for *nameTemplate* is:

```
fnb12345
```

If FNA12345 does not exist, the next name returned is again:

```
fna12345
```

_mktemp_s can create a maximum of 26 unique file names for any given combination of *base* and *nameTemplate* values. Therefore, FNZ12345 is the last unique file name **_mktemp_s** can create for the *base* and *nameTemplate* values used in this example.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
_mktemp_s	<io.h>
_wmktemp_s	<io.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mktemp_s.cpp
/* The program uses _mktemp to create
* five unique filenames. It opens each filename
* to ensure that the next name is unique.
*/

#include <io.h>
#include <string.h>
#include <stdio.h>

char *fnTemplate = "fnXXXXXX";
char names[5][9];

int main()
{
   int i, err, sizeInChars;
   FILE *fp;

   for( i = 0; i < 5; i++ )
   {
      strcpy_s( names[i], sizeof(names[i]), fnTemplate );
      /* Get the size of the string and add one for the null terminator.*/
      sizeInChars = strnlen(names[i], 9) + 1;
      /* Attempt to find a unique filename: */
      err = _mktemp_s( names[i], sizeInChars );
      if( err != 0 )
         printf( "Problem creating the template" );
      else
      {
         if( fopen_s( &fp, names[i], "w" ) == 0 )
            printf( "Unique filename is %s\n", names[i] );
         else
            printf( "Cannot open %s\n", names[i] );
         fclose( fp );
      }
   }

   return 0;
}
```

Sample Output

```
Unique filename is fna03188
Unique filename is fnb03188
Unique filename is fnc03188
Unique filename is fnd03188
Unique filename is fne03188
```

See also

File Handling

fopen, _wfopen
_getmbcp
_getpid
_open, _wopen
_setmbcp
_tempnam, _wtempnam, tmpnam, _wtmpnam
tmpfile_s

mktime, _mktime32, _mktime64

10/31/2018 • 3 minutes to read • Edit Online

Convert the local time to a calendar value.

Syntax

```
time_t mktime(
   struct tm *timeptr
);
__time32_t _mktime32(
   struct tm *timeptr
);
__time64_t _mktime64(
   struct tm *timeptr
);
```

Parameters

timeptr
Pointer to time structure; see asctime.

Return Value

_mktime32 returns the specified calendar time encoded as a value of type time_t. If *timeptr* references a date before midnight, January 1, 1970, or if the calendar time cannot be represented, **_mktime32** returns -1 cast to type **time_t**. When using **_mktime32** and if *timeptr* references a date after 23:59:59 January 18, 2038, Coordinated Universal Time (UTC), it will return -1 cast to type **time_t**.

_mktime64 will return -1 cast to type **__time64_t** if *timeptr* references a date after 23:59:59, December 31, 3000, UTC.

Remarks

The **mktime**, **_mktime32** and **_mktime64** functions convert the supplied time structure (possibly incomplete) pointed to by *timeptr* into a fully defined structure with normalized values and then converts it to a **time_t** calendar time value. The converted time has the same encoding as the values returned by the time function. The original values of the **tm_wday** and **tm_yday** components of the *timeptr* structure are ignored, and the original values of the other components are not restricted to their normal ranges.

mktime is an inline function that is equivalent to **_mktime64**, unless **_USE_32BIT_TIME_T** is defined, in which case it is equivalent to **_mktime32**.

After an adjustment to UTC, **_mktime32** handles dates from midnight, January 1, 1970, to 23:59:59 January 18, 2038, UTC. **_mktime64** handles dates from midnight, January 1, 1970 to 23:59:59, December 31, 3000. This adjustment may cause these functions to return -1 (cast to **time_t**, **__time32_t** or **__time64_t**) even though the date you specify is within range. For example, if you are in Cairo, Egypt, which is two hours ahead of UTC, two hours will first be subtracted from the date you specify in *timeptr*; this may now put your date out of range.

These functions may be used to validate and fill in a tm structure. If successful, these functions set the values of **tm_wday** and **tm_yday** as appropriate and set the other components to represent the specified calendar time, but with their values forced to the normal ranges. The final value of **tm_mday** is not set until **tm_mon** and

tm_year are determined. When specifying a **tm** structure time, set the **tm_isdst** field to:

- Zero (0) to indicate that standard time is in effect.
- A value greater than 0 to indicate that daylight saving time is in effect.
- A value less than zero to have the C run-time library code compute whether standard time or daylight saving time is in effect.

The C run-time library will determine the daylight savings time behavior from the TZ environment variable. If **TZ** is not set, the Win32 API call GetTimeZoneInformation is used to get the daylight savings time information from the operating system. If this fails, the library assumes the United States' rules for implementing the calculation of daylight saving time are used. **tm_isdst** is a required field. If not set, its value is undefined and the return value from these functions is unpredictable. If *timeptr* points to a **tm** structure returned by a previous call to asctime, gmtime, or localtime (or variants of these functions), the **tm_isdst** field contains the correct value.

Note that **gmtime** and **localtime** (and **_gmtime32**, **_gmtime64**, **_localtime32**, and **_localtime64**) use a single buffer per thread for the conversion. If you supply this buffer to **mktime**, **_mktime32** or **_mktime64**, the previous contents are destroyed.

These functions validate their parameter. If *timeptr* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
mktime	<time.h>
_mktime32	<time.h>
_mktime64	<time.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_mktime.c
/* The example takes a number of days
* as input and returns the time, the current
* date, and the specified number of days.
*/

#include <time.h>
#include <stdio.h>

int main( void )
{
   struct tm  when;
   __time64_t now, result;
   int        days;
   char       buff[80];

   time( &now );
   _localtime64_s( &when, &now );
   asctime_s( buff, sizeof(buff), &when );
   printf( "Current time is %s\n", buff );
   days = 20;
   when.tm_mday = when.tm_mday + days;
   if( (result = mktime( &when )) != (time_t)-1 ) {
      asctime_s( buff, sizeof(buff), &when );
      printf( "In %d days the time will be %s\n", days, buff );
   } else
      perror( "mktime failed" );
}
```

Sample Output

```
Current time is Fri Apr 25 13:34:07 2003

In 20 days the time will be Thu May 15 13:34:07 2003
```

See also

Time Management
asctime, _wasctime
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
_mkgmtime, _mkgmtime32, _mkgmtime64
time, _time32, _time64

modf, modff, modfl

10/31/2018 • 2 minutes to read • Edit Online

Splits a floating-point value into fractional and integer parts.

Syntax

```
double modf( double x, double * intptr );
float modff( float x, float * intptr );
long double modfl( long double x, long double * intptr );
```

```
float modf( float x, float * intptr );  // C++ only
long double modf( long double x, long double * intptr );  // C++ only
```

Parameters

x
Floating-point value.

intptr
Pointer to stored integer portion.

Return Value

This function returns the signed fractional portion of *x*. There is no error return.

Remarks

The **modf** functions break down the floating-point value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

modf has an implementation that uses Streaming SIMD Extensions 2 (SSE2). See _set_SSE2_enable for information and restrictions on using the SSE2 implementation.

C++ allows overloading, so you can call overloads of **modf** that take and return **float** or **long double** parameters. In a C program, **modf** always takes two double values and returns a double value.

Requirements

ROUTINE	REQUIRED HEADER
modf, **modff**, **modfl**	C: <math.h> C++: , <cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_modf.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x, y, n;

   x = -14.87654321;      /* Divide x into its fractional */
   y = modf( x, &n );     /* and integer parts            */

   printf( "For %f, the fraction is %f and the integer is %.f\n",
           x, y, n );
}
```

```
For -14.876543, the fraction is -0.876543 and the integer is -14
```

See also

Floating-Point Support
frexp
ldexp

_msize

10/31/2018 • 2 minutes to read • Edit Online

Returns the size of a memory block allocated in the heap.

Syntax

```
size_t _msize(
   void *memblock
);
```

Parameters

memblock
Pointer to the memory block.

Return Value

_msize returns the size (in bytes) as an unsigned integer.

Remarks

The **_msize** function returns the size, in bytes, of the memory block allocated by a call to **calloc**, **malloc**, or **realloc**.

When the application is linked with a debug version of the C run-time libraries, **_msize** resolves to _msize_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

This function validates its parameter. If *memblock* is a null pointer, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

Requirements

ROUTINE	REQUIRED HEADER
_msize	<malloc.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

See the example for realloc.

See also

Memory Allocation
calloc

_expand
malloc
realloc

_msize_dbg

10/31/2018 • 2 minutes to read • Edit Online

Calculates the size of a block of memory in the heap (debug version only).

Syntax

```
size_t _msize_dbg(
   void *userData,
   int blockType
);
```

Parameters

userData
Pointer to the memory block for which to determine the size.

blockType
Type of the specified memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

Return Value

On successful completion, **_msize_dbg** returns the size (in bytes) of the specified memory block; otherwise it returns **NULL**.

Remarks

_msize_dbg is a debug version of the _msize function. When _DEBUG is not defined, each call to **_msize_dbg** is reduced to a call to **_msize**. Both **_msize** and **_msize_dbg** calculate the size of a memory block in the base heap, but **_msize_dbg** adds two debugging features: It includes the buffers on either side of the user portion of the memory block in the returned size and it allows size calculations for specific block types.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

This function validates its parameter. If *memblock* is a null pointer, **_msize** invokes an invalid parameter handler, as described in Parameter Validation. If the error is handled, the function sets **errno** to **EINVAL** and returns -1.

Requirements

ROUTINE	REQUIRED HEADER
_msize_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

```
// crt_msize_dbg.c
// compile with: /MTd
/*
* This program allocates a block of memory using _malloc_dbg
* and then calls _msize_dbg to display the size of that block.
* Next, it uses _realloc_dbg to expand the amount of
* memory used by the buffer and then calls _msize_dbg again to
* display the new amount of memory allocated to the buffer.
*/

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <crtdbg.h>

int main( void )
{
        long *buffer, *newbuffer;
        size_t size;

        /*
         * Call _malloc_dbg to include the filename and line number
         * of our allocation request in the header
         */
        buffer = (long *)_malloc_dbg( 40 * sizeof(long), _NORMAL_BLOCK, __FILE__, __LINE__ );
        if( buffer == NULL )
               exit( 1 );

        /*
         * Get the size of the buffer by calling _msize_dbg
         */
        size = _msize_dbg( buffer, _NORMAL_BLOCK );
        printf( "Size of block after _malloc_dbg of 40 longs: %u\n", size );

        /*
         * Reallocate the buffer using _realloc_dbg and show the new size
         */
        newbuffer = _realloc_dbg( buffer, size + (40 * sizeof(long)), _NORMAL_BLOCK, __FILE__, __LINE__ );
        if( newbuffer == NULL )
               exit( 1 );
        buffer = newbuffer;
        size = _msize_dbg( buffer, _NORMAL_BLOCK );
        printf( "Size of block after _realloc_dbg of 40 more longs: %u\n", size );

        free( buffer );
        exit( 0 );
}
```

Output

```
Size of block after _malloc_dbg of 40 longs: 160
Size of block after _realloc_dbg of 40 more longs: 320
```

See also

Debug Routines
_malloc_dbg

nan, nanf, nanl

2/4/2019 • 2 minutes to read • Edit Online

Returns a quiet NaN value.

Syntax

```
double nan( const char* input );
float nanf( const char* input );
long double nanl( const char* input );
```

Parameters

input
A string value.

Return Value

The **nan** functions return a quiet NaN value.

Remarks

The **nan** functions return a floating-point value that corresponds to a quiet (non-signalling) NaN. The *input* value is ignored. For information about how a NaN is represented for output, see printf, _printf_l, wprintf, _wprintf_l.

Requirements

FUNCTION	C HEADER	C++ HEADER
nan, **nanf**, **nanl**	<math.h>	<cmath> or <math.h>

See also

Floating-Point Support
fpclassify
_fpclass, _fpclassf
isfinite, _finite, _finitef
isinf
isnan, _isnan, _isnanf
isnormal

nearbyint, nearbyintf, nearbyintl

2/4/2019 • 2 minutes to read • Edit Online

Rounds the specified floating-point value to an integer, and returns that value in a floating-point format.

Syntax

```
double nearbyint( double x );
float nearbyintf( float x );
long double nearbyintl( long double x );
```

```
float nearbyint( float x ); //C++ only
long double nearbyint( long double x ); //C++ only
```

Parameters

x
The value to round.

Return Value

If successful, returns *x*, rounded to the nearest integer, using the current rounding format as reported by fegetround. Otherwise, the function may return one of the following values:

ISSUE	RETURN
x = ±INFINITY	±INFINITY, unmodified
x = ±0	±0, unmodified
x = NaN	NaN

Errors are not reported through _matherr; specifically, this function does not report any **FE_INEXACT** exceptions.

Remarks

The primary difference between this function and rint is that this function does not raise the inexact floating point exception.

Because the maximum floating-point values are exact integers, this function will never overflow by itself; rather, the output may overflow the return value, depending on which version of the function you use.

C++ allows overloading, so you can call overloads of **nearbyint** that take and return **float** or **long double** parameters. In a C program, **nearbyint** always takes two double values and returns a double value.

Requirements

FUNCTION	C HEADER	C++ HEADER
nearbyint, **nearbyintf**, **nearbyintl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
Math and floating-point support

nextafter, nextafterf, nextafterl, _nextafter, _nextafterf, nexttoward, nexttowardf, nexttowardl

10/31/2018 • 2 minutes to read • Edit Online

Returns the next representable floating-point value.

Syntax

```
double nextafter( double x, double y );
float nextafterf( float x, float y );
long double nextafterl( long double x, long double y );

double _nextafter( double x, double y );
float _nextafterf( float x, float y ); /* x64 only */

double nexttoward( double x, long double y );
float nexttowardf( float x, long double y );
long double nexttowardl( long double x, long double y );
```

```
float nextafter( float x, float y ); /* C++ only, requires <cmath> */
long double nextafter( long double x, long double y ); /* C++ only, requires <cmath> */

float nexttoward( float x, long double y ); /* C++ only, requires <cmath> */
long double nexttoward( long double x, long double y ); /* C++ only, requires <cmath> */
```

Parameters

x
The floating-point value to start from.

y
The floating-point value to go towards.

Return Value

Returns the next representable floating-point value of the return type after *x* in the direction of *y*. If *x* and *y* are equal, the function returns *y*, converted to the return type, with no exception triggered. If *x* is not equal to *y*, and the result is a denormal or zero, the **FE_UNDERFLOW** and **FE_INEXACT** floating-point exception states are set, and the correct result is returned. If either *x* or *y* is a NAN, then the return value is one of the input NANs. If *x* is finite and the result is infinite or not representable in the type, a correctly signed infinity or NAN is returned, the **FE_OVERFLOW** and **FE_INEXACT** floating-point exception states are set, and **errno** is set to **ERANGE**.

Remarks

The **nextafter** and **nexttoward** function families are equivalent, except for the parameter type of *y*. If *x* and *y* are equal, the value returned is *y* converted to the return type.

Because C++ allows overloading, if you include <cmath> you can call overloads of **nextafter** and **nexttoward** that return **float** and **long double** types. In a C program, **nextafter** and **nexttoward** always return **double**.

The **_nextafter** and **_nextafterf** functions are Microsoft specific. The **_nextafterf** function is only available when compiling for x64.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
nextafter, **nextafterf**, **nextafterl**, **_nextafterf**, **nexttoward**, **nexttowardf**, **nexttowardl**	<math.h>	<math.h> or <cmath>
_nextafter	<float.h>	<float.h> or <cfloat>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
isnan, _isnan, _isnanf

norm, normf, norml

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the squared magnitude of a complex number.

Syntax

```
double norm( _Dcomplex z );
float normf( _Fcomplex z );
long double norml( _Lcomplex z );
```

```
float norm( _Fcomplex z );  // C++ only
long double norm( _Lcomplex z );  // C++ only
```

Parameters

z
A complex number.

Return Value

The squared magnitude of *z*.

Remarks

Because C++ allows overloading, you can call overloads of **norm** that take **_Fcomplex** or **_Lcomplex** values, and return **float** or **long double** values. In a C program, **norm** always takes a **_Dcomplex** value and returns a **double** value.

Requirements

ROUTINE	C HEADER	C++ HEADER
norm, **normf**, **norml**	<complex.h>	<complex.h>

The **_Fcomplex**, **_Dcomplex**, and **_Lcomplex** types are Microsoft-specific equivalents of the unimplemented native C99 types **float _Complex**, **double _Complex**, and **long double _Complex**, respectively. For more compatibility information, see Compatibility.

See also

Alphabetical Function Reference
creal, crealf, creall
cproj, cprojf, cprojl
conj, conjf, conjl
cimag, cimagf, cimagl
carg, cargf, cargl
cabs, cabsf, cabsl

not

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the ! operator.

Syntax

```
#define not !
```

Remarks

The macro yields the operator !.

Example

```
// iso646_not.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 0;

   if (!a)
      cout << "a is zero" << endl;

   if (not(a))
      cout << "a is zero" << endl;
}
```

```
a is zero
a is zero
```

Requirements

Header: <iso646.h>

not_eq

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the != operator.

Syntax

```
#define not_eq !=
```

Remarks

The macro yields the operator !=.

Example

```
// iso646_not_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 0, b = 1;

   if (a != b)
      cout << "a is not equal to b" << endl;

   if (a not_eq b)
      cout << "a is not equal to b" << endl;
}
```

```
a is not equal to b
a is not equal to b
```

Requirements

Header: <iso646.h>

offsetof Macro

10/31/2018 • 2 minutes to read • Edit Online

Retrieves the offset of a member from the beginning of its parent structure.

Syntax

```
size_t offsetof(
   structName,
   memberName
);
```

Parameters

structName
Name of the parent data structure.

memberName
Name of the member in the parent data structure for which to determine the offset.

Return Value

offsetof returns the offset in bytes of the specified member from the beginning of its parent data structure. It is undefined for bit fields.

Remarks

The **offsetof** macro returns the offset in bytes of *memberName* from the beginning of the structure specified by *structName* as a value of type **size_t**. You can specify types with the **struct** keyword.

> **NOTE**
> **offsetof** is not a function and cannot be described using a C prototype.

Requirements

ROUTINE	REQUIRED HEADER
offsetof	<stddef.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Memory Allocation

_onexit, _onexit_m

10/31/2018 • 2 minutes to read • Edit Online

Registers a routine to be called at exit time.

Syntax

```
_onexit_t _onexit(
   _onexit_t function
);
_onexit_t_m _onexit_m(
   _onexit_t_m function
);
```

Parameters

function
Pointer to a function to be called at exit.

Return Value

_onexit returns a pointer to the function if successful or **NULL** if there is no space to store the function pointer.

Remarks

The **_onexit** function is passed the address of a function (*function*) to be called when the program terminates normally. Successive calls to **_onexit** create a register of functions that are executed in LIFO (last-in-first-out) order. The functions passed to **_onexit** cannot take parameters.

In the case when **_onexit** is called from within a DLL, routines registered with **_onexit** run on a DLL's unloading after **DllMain** is called with DLL_PROCESS_DETACH.

_onexit is a Microsoft extension. For ANSI portability, use atexit. The **_onexit_m** version of the function is for mixed mode use.

Requirements

ROUTINE	REQUIRED HEADER
_onexit	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_onexit.c

#include <stdlib.h>
#include <stdio.h>

/* Prototypes */
int fn1(void), fn2(void), fn3(void), fn4 (void);

int main( void )
{
   _onexit( fn1 );
   _onexit( fn2 );
   _onexit( fn3 );
   _onexit( fn4 );
   printf( "This is executed first.\n" );
}

int fn1()
{
   printf( "next.\n" );
   return 0;
}

int fn2()
{
   printf( "executed " );
   return 0;
}

int fn3()
{
   printf( "is " );
   return 0;
}

int fn4()
{
   printf( "This " );
   return 0;
}
```

Output

```
This is executed first.
This is executed next.
```

See also

Process and Environment Control
atexit
exit, _Exit, _exit
__dllonexit

open

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _open instead.

_open, _wopen

11/8/2018 • 6 minutes to read • Edit Online

Opens a file. These functions are deprecated because more-secure versions are available; see _sopen_s, _wsopen_s.

Syntax

```
int _open(
   const char *filename,
   int oflag [,
   int pmode]
);
int _wopen(
   const wchar_t *filename,
   int oflag [,
   int pmode]
);
```

Parameters

filename
File name.

oflag
The kind of operations allowed.

pmode
Permission mode.

Return Value

Each of these functions returns a file descriptor for the opened file. A return value of -1 indicates an error; in that case **errno** is set to one of the following values.

ERRNO VALUE	CONDITION
EACCES	Tried to open a read-only file for writing, file's sharing mode does not allow the specified operations, or the given path is a directory.
EEXIST	**_O_CREAT** and **_O_EXCL** flags specified, but *filename* already exists.
EINVAL	Invalid *oflag* or *pmode* argument.
EMFILE	No more file descriptors are available (too many files are open).
ENOENT	File or path not found.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_open** function opens the file specified by *filename* and prepares it for reading or writing, as specified by *oflag*. **_wopen** is a wide-character version of **_open**; the *filename* argument to **_wopen** is a wide-character string. **_wopen** and **_open** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_topen	**_open**	**_open**	**_wopen**

oflag is an integer expression formed from one or more of the following manifest constants or constant combinations, which are defined in <fcntl.h>.

OFLAG CONSTANT	BEHAVIOR
_O_APPEND	Moves the file pointer to the end of the file before every write operation.
_O_BINARY	Opens the file in binary (untranslated) mode. (See fopen for a description of binary mode.)
_O_CREAT	Creates a file and opens it for writing. Has no effect if the file specified by *filename* exists. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_SHORT_LIVED**	Creates a file as temporary and if possible does not flush to disk. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_TEMPORARY**	Creates a file as temporary; the file is deleted when the last file descriptor is closed. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | `_O_EXCL`	Returns an error value if a file specified by *filename* exists. Applies only when used with **_O_CREAT**.
_O_NOINHERIT	Prevents creation of a shared file descriptor.
_O_RANDOM	Specifies that caching is optimized for, but not restricted to, random access from disk.
_O_RDONLY	Opens a file for reading only. Cannot be specified with **_O_RDWR** or **_O_WRONLY**.
_O_RDWR	Opens a file for both reading and writing. Cannot be specified with **_O_RDONLY** or **_O_WRONLY**.
_O_SEQUENTIAL	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
_O_TEXT	Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)

OFLAG CONSTANT	BEHAVIOR
_O_TRUNC	Opens a file and truncates it to zero length; the file must have write permission. Cannot be specified with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a file. **Note:** The **_O_TRUNC** flag destroys the contents of the specified file.
_O_WRONLY	Opens a file for writing only. Cannot be specified with **_O_RDONLY** or **_O_RDWR**.
_O_U16TEXT	Opens a file in Unicode UTF-16 mode.
_O_U8TEXT	Opens a file in Unicode UTF-8 mode.
_O_WTEXT	Opens a file in Unicode mode.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value for the access mode.

If **_O_WTEXT** is used to open a file for reading, **_open** reads the beginning of the file and checks for a byte order mark (BOM). If there is a BOM, the file is treated as UTF-8 or UTF-16LE, depending on the BOM. If no BOM is present, the file is treated as ANSI. When a file is opened for writing by using **_O_WTEXT**, UTF-16 is used. Regardless of any previous setting or byte order mark, if **_O_U8TEXT** is used, the file is always opened as UTF-8; if **_O_U16TEXT** is used, the file is always opened as UTF-16.

When a file is opened in Unicode mode by using **_O_WTEXT**, **_O_U8TEXT**, or **_O_U16TEXT**, input functions translate the data that's read from the file into UTF-16 data stored as type **wchar_t**. Functions that write to a file opened in Unicode mode expect buffers that contain UTF-16 data stored as type **wchar_t**. If the file is encoded as UTF-8, then UTF-16 data is translated into UTF-8 when it is written, and the file's UTF-8-encoded content is translated into UTF-16 when it is read. An attempt to read or write an odd number of bytes in Unicode mode causes a parameter validation error. To read or write data that's stored in your program as UTF-8, use a text or binary file mode instead of a Unicode mode. You are responsible for any required encoding translation.

If **_open** is called with **_O_WRONLY** | **_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it first tries to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it opens the file for writing only and uses the default value for the Unicode mode setting.

When two or more manifest constants are used to form the *oflag* argument, the constants are combined with the bitwise-OR operator (**|**). For a discussion of binary and text modes, see Text and Binary Mode File I/O.

The *pmode* argument is required only when **_O_CREAT** is specified. If the file already exists, *pmode* is ignored. Otherwise, *pmode* specifies the file permission settings, which are set when the new file is closed the first time. **_open** applies the current file-permission mask to *pmode* before the permissions are set. (For more information, see _umask.) *pmode* is an integer expression that contains one or both of the following manifest constants, which are defined in <sys\stat.h>.

PMODE	MEANING
_S_IREAD	Only reading permitted.

PMODE	MEANING
_S_IWRITE	Writing permitted. (In effect, permits reading and writing.)
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). In Windows, all files are readable; write-only permission is not available. Therefore, the modes **_S_IWRITE** and **_S_IREAD** | **_S_IWRITE** are equivalent.

If a value other than some combination of **_S_IREAD** and **_S_IWRITE** is specified for *pmode*—even if it would specify a valid *pmode* in another operating system—or if any value other than the allowed *oflag* values is specified, the function generates an assertion in Debug mode and invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_open	<io.h>	<fcntl.h>, <sys\types.h>, <sys\stat.h>
_wopen	<io.h> or <wchar.h>	<fcntl.h>, <sys\types.h>, <sys\stat.h>

_open and **_wopen** are Microsoft extensions. For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_open.c
// compile with: /W3
/* This program uses _open to open a file
* named CRT_OPEN.C for input and a file named CRT_OPEN.OUT
* for output. The files are then closed.
*/
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>

int main( void )
{
   int fh1, fh2;

   fh1 = _open( "CRT_OPEN.C", _O_RDONLY ); // C4996
   // Note: _open is deprecated; consider using _sopen_s instead
   if( fh1 == -1 )
      perror( "Open failed on input file" );
   else
   {
      printf( "Open succeeded on input file\n" );
      _close( fh1 );
   }

   fh2 = _open( "CRT_OPEN.OUT", _O_WRONLY | _O_CREAT, _S_IREAD |
                           _S_IWRITE ); // C4996
   if( fh2 == -1 )
      perror( "Open failed on output file" );
   else
   {
      printf( "Open succeeded on output file\n" );
      _close( fh2 );
   }
}
```

Output

```
Open succeeded on input file
Open succeeded on output file
```

See also

Low-Level I/O
_chmod, _wchmod
_close
_creat, _wcreat
_dup, _dup2
fopen, _wfopen
_sopen, _wsopen

_open_osfhandle

5/23/2019 • 2 minutes to read • Edit Online

Associates a C run-time file descriptor with an existing operating-system file handle.

Syntax

```
int _open_osfhandle (
   intptr_t osfhandle,
   int flags
);
```

Parameters

osfhandle
Operating-system file handle.

flags
Types of operations allowed.

Return Value

If successful, **_open_osfhandle** returns a C run-time file descriptor. Otherwise, it returns -1.

Remarks

The **_open_osfhandle** function allocates a C run-time file descriptor and associates it with the operating-system file handle specified by *osfhandle*. To avoid a compiler warning, cast the *osfhandle* argument from **HANDLE** to **intptr_t**. The *flags* argument is an integer expression formed from one or more of the manifest constants defined in <fcntl.h>. When two or more manifest constants are used to form the *flags* argument, the constants are combined with the bitwise-OR operator (**|**).

These manifest constants are defined in <fcntl.h>:

_O_APPEND	Positions a file pointer to the end of the file before every write operation.
_O_RDONLY	Opens the file for reading only.
_O_TEXT	Opens the file in text (translated) mode.
_O_WTEXT	Opens the file in Unicode (translated UTF-16) mode.

The **_open_osfhandle** call transfers ownership of the Win32 file handle to the file descriptor. To close a file opened by using **_open_osfhandle**, call _close. The underlying OS file handle is also closed by a call to **_close**. Don't call the Win32 function **CloseHandle** on the original handle. If the file descriptor is owned by a **FILE *** stream, then a call to fclose on that **FILE *** stream closes both the file descriptor and the underlying handle. In this case, don't call **_close** on the file descriptor or **CloseHandle** on the original handle.

Requirements

ROUTINE	REQUIRED HEADER
_open_osfhandle	<io.h>

For more compatibility information, see Compatibility.

See also

File Handling

or_eq

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the |= operator.

Syntax

```
#define or_eq |=
```

Remarks

The macro yields the operator |=.

Example

```
// iso646_oreq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a |= b;
   cout << result << endl;

   result= a or_eq b;
   cout << result << endl;
}
```

```
3
3
```

Requirements

Header: <iso646.h>

or

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the || operator.

Syntax

```
#define or ||
```

Remarks

The macro yields the operator ||.

Example

```
// iso646_or.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   bool a = true, b = false, result;

   boolalpha(cout);

   result= a || b;
   cout << result << endl;

   result= a or b;
   cout << result << endl;
}
```

```
true
true
```

Requirements

Header: <iso646.h>

_pclose

10/31/2018 • 2 minutes to read • Edit Online

Waits for a new command processor and closes the stream on the associated pipe.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _pclose(
FILE *stream
);
```

Parameters

stream
Return value from the previous call to **_popen**.

Return Value

Returns the exit status of the terminating command processor, or -1 if an error occurs. The format of the return value is the same as that for **_cwait**, except the low-order and high-order bytes are swapped. If stream is **NULL**, **_pclose** sets **errno** to **EINVAL** and returns -1.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_pclose** function looks up the process ID of the command processor (Cmd.exe) started by the associated **_popen** call, executes a _cwait call on the new command processor, and closes the stream on the associated pipe.

Requirements

ROUTINE	REQUIRED HEADER
_pclose	<stdio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Process and Environment Control
_pipe

_popen, _wpopen

perror, _wperror

10/31/2018 • 2 minutes to read • Edit Online

Print an error message.

Syntax

```
void perror(
   const char *message
);
void _wperror(
   const wchar_t *message
);
```

Parameters

message
String message to print.

Remarks

The **perror** function prints an error message to **stderr**. **_wperror** is a wide-character version of **_perror**; the *message* argument to **_wperror** is a wide-character string. **_wperror** and **_perror** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tperror	**perror**	**perror**	**_wperror**

message is printed first, followed by a colon, then by the system error message for the last library call that produced the error, and finally by a newline character. If *message* is a null pointer or a pointer to a null string, **perror** prints only the system error message.

The error number is stored in the variable errno (defined in ERRNO.H). The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. **perror** prints the appropriate error message using the **errno** value as an index to **_sys_errlist**. The value of the variable _sys_nerr is defined as the maximum number of elements in the **_sys_errlist** array.

For accurate results, call **perror** immediately after a library routine returns with an error. Otherwise, subsequent calls can overwrite the **errno** value.

In the Windows operating system, some **errno** values listed in ERRNO.H are unused. These values are reserved for use by the UNIX operating system. See _doserrno, errno, _sys_errlist, and _sys_nerr for a listing of **errno** values used by the Windows operating system. **perror** prints an empty string for any **errno** value not used by these platforms.

Requirements

ROUTINE	REQUIRED HEADER
perror	<stdio.h> or <stdlib.h>
_wperror	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_perror.c
// compile with: /W3
/* This program attempts to open a file named
* NOSUCHF.ILE. Because this file probably doesn't exist,
* an error message is displayed. The same message is
* created using perror, strerror, and _strerror.
*/

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <share.h>

int main( void )
{
   int  fh;

   if( _sopen_s( &fh, "NOSUCHF.ILE", _O_RDONLY, _SH_DENYNO, 0 ) != 0 )
   {
      /* Three ways to create error message: */
      perror( "perror says open failed" );
      printf( "strerror says open failed: %s\n",
         strerror( errno ) ); // C4996
      printf( _strerror( "_strerror says open failed" ) ); // C4996
      // Note: strerror and _strerror are deprecated; consider
      // using strerror_s and _strerror_s instead.
   }
   else
   {
      printf( "open succeeded on input file\n" );
      _close( fh );
   }
}
```

```
perror says open failed: No such file or directory
strerror says open failed: No such file or directory
_strerror says open failed: No such file or directory
```

See also

Process and Environment Control

clearerr

ferror

strerror, _strerror, _wcserror, __wcserror

_pipe

10/31/2018 • 7 minutes to read • Edit Online

Creates a pipe for reading and writing.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _pipe(
   int *pfds,
   unsigned int psize,
   int textmode
);
```

Parameters

pfds
Pointer to an array of two **int** to hold read and write file descriptors.

psize
Amount of memory to reserve.

textmode
File mode.

Return Value

Returns 0 if successful. Returns -1 to indicate an error. On error,**errno** is set to one of these values:

- **EMFILE**, which indicates that no more file descriptors are available.
- **ENFILE**, which indicates a system-file-table overflow.
- **EINVAL**, which indicates that either the array *pfds* is a null pointer or that an invalid value for *textmode* was passed in.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **_pipe** function creates a *pipe*, which is an artificial I/O channel that a program uses to pass information to other programs. A pipe resembles a file because it has a file pointer, a file descriptor, or both, and it can be read from or written to by using the Standard Library input and output functions. However, a pipe does not represent a specific file or device. Instead, it represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the operating system.

_pipe resembles **_open** but opens the pipe for reading and writing and returns two file descriptors instead of one. The program can use both sides of the pipe or close the one that it doesn't need. For example, the command

processor in Windows creates a pipe when it executes a command such as **PROGRAM1** | **PROGRAM2**.

The standard output descriptor of **PROGRAM1** is attached to the pipe's write descriptor. The standard input descriptor of **PROGRAM2** is attached to the pipe's read descriptor. This eliminates the need to create temporary files to pass information to other programs.

The **_pipe** function returns two file descriptors to the pipe in the *pfds* argument. The element *pfds*[0] contains the read descriptor, and the element *pfds*[1] contains the write descriptor. Pipe file descriptors are used in the same way as other file descriptors. (The low-level input and output functions **_read** and **_write** can read from and write to a pipe.) To detect the end-of-pipe condition, check for a **_read** request that returns 0 as the number of bytes read.

The *psize* argument specifies the amount of memory, in bytes, to reserve for the pipe. The *textmode* argument specifies the translation mode for the pipe. The manifest constant **_O_TEXT** specifies a text translation, and the constant **_O_BINARY** specifies binary translation. (See fopen, _wfopen for a description of text and binary modes.) If the *textmode* argument is 0, **_pipe** uses the default translation mode that's specified by the default-mode variable _fmode.

In multithreaded programs, no locking is performed. The file descriptors that are returned are newly opened and should not be referenced by any thread until after the **_pipe** call is complete.

To use the **_pipe** function to communicate between a parent process and a child process, each process must have only one descriptor open on the pipe. The descriptors must be opposites: if the parent has a read descriptor open, then the child must have a write descriptor open. The easiest way to do this is to bitwise or (**|**) the **_O_NOINHERIT** flag with *textmode*. Then, use **_dup** or **_dup2** to create an inheritable copy of the pipe descriptor that you want to pass to the child. Close the original descriptor, and then spawn the child process. On returning from the spawn call, close the duplicate descriptor in the parent process. For more information, see example 2 later in this article.

In the Windows operating system, a pipe is destroyed when all of its descriptors have been closed. (If all read descriptors on the pipe have been closed, then writing to the pipe causes an error.) All read and write operations on the pipe wait until there is enough data or enough buffer space to complete the I/O request.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_pipe	<io.h>	<fcntl.h>,1 <errno.h>2

1 For **_O_BINARY** and **_O_TEXT** definitions.

2 **errno** definitions.

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example 1

```
// crt_pipe.c
/* This program uses the _pipe function to pass streams of
* text to spawned processes.
*/

#include <stdlib.h>
```

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <math.h>

enum PIPES { READ, WRITE }; /* Constants 0 and 1 for READ and WRITE */
#define NUMPROBLEM 8

int main( int argc, char *argv[] )
{

   int fdpipe[2];
   char hstr[20];
   int pid, problem, c;
   int termstat;

   /* If no arguments, this is the spawning process */
   if( argc == 1 )
   {

      setvbuf( stdout, NULL, _IONBF, 0 );

      /* Open a set of pipes */
      if( _pipe( fdpipe, 256, O_BINARY ) == -1 )
          exit( 1 );

      /* Convert pipe read descriptor to string and pass as argument
       * to spawned program. Program spawns itself (argv[0]).
       */
      _itoa_s( fdpipe[READ], hstr, sizeof(hstr), 10 );
      if( ( pid = _spawnl( P_NOWAIT, argv[0], argv[0],
            hstr, NULL ) ) == -1 )
          printf( "Spawn failed" );

      /* Put problem in write pipe. Since spawned program is
       * running simultaneously, first solutions may be done
       * before last problem is given.
       */
      for( problem = 1000; problem <= NUMPROBLEM * 1000; problem += 1000)
      {

         printf( "Son, what is the square root of %d?\n", problem );
         _write( fdpipe[WRITE], (char *)&problem, sizeof( int ) );

      }

      /* Wait until spawned program is done processing. */
      _cwait( &termstat, pid, WAIT_CHILD );
      if( termstat & 0x0 )
         printf( "Child failed\n" );

      _close( fdpipe[READ] );
      _close( fdpipe[WRITE] );

   }

   /* If there is an argument, this must be the spawned process. */
   else
   {

      /* Convert passed string descriptor to integer descriptor. */
      fdpipe[READ] = atoi( argv[1] );

      /* Read problem from pipe and calculate solution. */
      for( c = 0; c < NUMPROBLEM; c++ )
      {

        _read( fdpipe[READ], (char *)&problem, sizeof( int ) );
```

```
        printf( "Dad, the square root of %d is %3.2f.\n",
                 problem, sqrt( ( double )problem ) );

      }
   }
}
```

```
Son, what is the square root of 1000?
Son, what is the square root of 2000?
Son, what iDad, the square root of 1000 is 31.62.
Dad, the square root of 2000 is 44.72.
s the square root of 3000?
Dad, the square root of 3000 is 54.77.
Son, what is the square root of 4000?
Dad, the square root of 4000 is 63.25.
Son, what is the square root of 5000?
Dad, the square root of 5000 is 70.71.
Son, what is the square root of 6000?
SonDad, the square root of 6000 is 77.46.
, what is the square root of 7000?
Dad, the square root of 7000 is 83.67.
Son, what is the square root of 8000?
Dad, the square root of 8000 is 89.44.
```

Example 2

This is a basic filter application. It spawns the application crt_pipe_beeper after it creates a pipe that directs the spawned application's stdout to the filter. The filter removes ASCII 7 (beep) characters.

```
// crt_pipe_beeper.c

#include <stdio.h>
#include <string.h>

int main()
{
   int   i;
   for(i=0;i<10;++i)
      {
         printf("This is speaker beep number %d...\n\7", i+1);
      }
   return 0;
}
```

The actual filter application:

```
// crt_pipe_BeepFilter.C
// arguments: crt_pipe_beeper.exe

#include <windows.h>
#include <process.h>
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

#define   OUT_BUFF_SIZE 512
#define   READ_FD 0
#define   WRITE_FD 1
#define   BEEP_CHAR 7
```

```
char szBuffer[OUT_BUFF_SIZE];

int Filter(char* szBuff, ULONG nSize, int nChar)
{
   char* szPos = szBuff + nSize -1;
   char* szEnd = szPos;
   int nRet = nSize;

   while (szPos > szBuff)
   {
      if (*szPos == nChar)
         {
            memmove(szPos, szPos+1, szEnd - szPos);
            --nRet;
         }
      --szPos;
   }
   return nRet;
}

int main(int argc, char** argv)
{
   int nExitCode = STILL_ACTIVE;
   if (argc >= 2)
   {
      HANDLE hProcess;
      int fdStdOut;
      int fdStdOutPipe[2];

      // Create the pipe
      if(_pipe(fdStdOutPipe, 512, O_NOINHERIT) == -1)
         return   1;

      // Duplicate stdout file descriptor (next line will close original)
      fdStdOut = _dup(_fileno(stdout));

      // Duplicate write end of pipe to stdout file descriptor
      if(_dup2(fdStdOutPipe[WRITE_FD], _fileno(stdout)) != 0)
         return   2;

      // Close original write end of pipe
      _close(fdStdOutPipe[WRITE_FD]);

      // Spawn process
      hProcess = (HANDLE)_spawnvp(P_NOWAIT, argv[1],
       (const char* const*)&argv[1]);

      // Duplicate copy of original stdout back into stdout
      if(_dup2(fdStdOut, _fileno(stdout)) != 0)
         return   3;

      // Close duplicate copy of original stdout
      _close(fdStdOut);

      if(hProcess)
      {
         int nOutRead;
         while   (nExitCode == STILL_ACTIVE)
         {
            nOutRead = _read(fdStdOutPipe[READ_FD],
             szBuffer, OUT_BUFF_SIZE);
            if(nOutRead)
            {
               nOutRead = Filter(szBuffer, nOutRead, BEEP_CHAR);
               fwrite(szBuffer, 1, nOutRead, stdout);
            }

            if(!GetExitCodeProcess(hProcess,(unsigned long*)&nExitCode))
               return 4;
```

```
        }
      }
   }
   return nExitCode;
}
```

```
This is speaker beep number 1...
This is speaker beep number 2...
This is speaker beep number 3...
This is speaker beep number 4...
This is speaker beep number 5...
This is speaker beep number 6...
This is speaker beep number 7...
This is speaker beep number 8...
This is speaker beep number 9...
This is speaker beep number 10...
```

See also

Process and Environment Control

_open, _wopen

_popen, _wpopen

10/31/2018 • 2 minutes to read • Edit Online

Creates a pipe and executes a command.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
FILE *_popen(
    const char *command,
    const char *mode
);
FILE *_wpopen(
    const wchar_t *command,
    const wchar_t *mode
);
```

Parameters

command
Command to be executed.

mode
Mode of the returned stream.

Return Value

Returns a stream associated with one end of the created pipe. The other end of the pipe is associated with the spawned command's standard input or standard output. The functions return **NULL** on an error. If the error is an invalid parameter, such as if *command* or *mode* is a null pointer, or *mode* is not a valid mode, **errno** is set to **EINVAL**. See the Remarks section for valid modes.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_popen** function creates a pipe and asynchronously executes a spawned copy of the command processor with the specified string *command*. The character string *mode* specifies the type of access requested, as follows.

ACCESS MODE	DESCRIPTION
"r"	The calling process can read the spawned command's standard output using the returned stream.
"w"	The calling process can write to the spawned command's standard input using the returned stream.

ACCESS MODE	DESCRIPTION
"b"	Open in binary mode.
"t"	Open in text mode.

> **NOTE**
> If used in a Windows program, the **_popen** function returns an invalid file pointer that causes the program to stop responding indefinitely. **_popen** works properly in a console application. To create a Windows application that redirects input and output, see Creating a Child Process with Redirected Input and Output in the Windows SDK.

_wpopen is a wide-character version of **_popen**; the *path* argument to **_wpopen** is a wide-character string. **_wpopen** and **_popen** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tpopen	**_popen**	**_popen**	**_wpopen**

Requirements

ROUTINE	REQUIRED HEADER
_popen	<stdio.h>
_wpopen	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_popen.c
/* This program uses _popen and _pclose to receive a
* stream of text from a system process.
*/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{

   char   psBuffer[128];
   FILE   *pPipe;

        /* Run DIR so that it writes its output to a pipe. Open this
         * pipe with read text attribute so that we can read it
         * like a text file.
         */

   if( (pPipe = _popen( "dir *.c /on /p", "rt" )) == NULL )
      exit( 1 );

   /* Read pipe until end of file, or an error occurs. */

   while(fgets(psBuffer, 128, pPipe))
   {
      printf(psBuffer);
   }

   /* Close pipe and print return value of pPipe. */
   if (feof( pPipe))
   {
     printf( "\nProcess returned %d\n", _pclose( pPipe ) );
   }
   else
   {
     printf( "Error: Failed to read the pipe to the end.\n");
   }
}
```

Sample Output

This output assumes that there is only one file in the current directory with a .c file name extension.

```
Volume in drive C is CDRIVE
Volume Serial Number is 0E17-1702

Directory of D:\proj\console\test1

07/17/98  07:26p                   780 popen.c
               1 File(s)            780 bytes
                             86,597,632 bytes free

Process returned 0
```

See also

Process and Environment Control
_pclose
_pipe

pow, powf, powl

10/31/2018 • 2 minutes to read • Edit Online

Calculates *x* raised to the power of *y*.

Syntax

```
double pow( double x, double y );
float powf( float x, float y );
long double powl( long double x, long double y );
```

```
double pow( double x, int y );  // C++ only
float pow( float x, float y );  // C++ only
float pow( float x, int y );  // C++ only
long double pow( long double x, long double y );  // C++ only
long double pow( long double x, int y );  // C++ only
```

Parameters

x
Base.

y
Exponent.

Return Value

Returns the value of x^y. No error message is printed on overflow or underflow.

VALUES OF X AND Y	RETURN VALUE OF POW
x != 0.0 and *y* == 0.0	1
x == 0.0 and *y* == 0.0	1
x == 0.0 and *y* < 0	INF

Remarks

pow does not recognize integral floating-point values greater than 2^{64} (for example, 1.0E100).

pow has an implementation that uses Streaming SIMD Extensions 2 (SSE2). For information and restrictions about using the SSE2 implementation, see _set_SSE2_enable.

Because C++ allows overloading, you can call any of the various overloads of **pow**. In a C program, **pow** always takes two **double** values and returns a **double** value.

The `pow(int, int)` overload is no longer available. If you use this overload, the compiler may emit C2668. To avoid this problem, cast the first parameter to **double**, **float**, or **long double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
pow, **powf**, **powl**	<math.h>	<math.h> or <cmath>

For additional compatibility information, see Compatibility.

Example

```
// crt_pow.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.0, y = 3.0, z;

   z = pow( x, y );
   printf( "%.1f to the power of %.1f is %.1f\n", x, y, z );
}
```

```
2.0 to the power of 3.0 is 8.0
```

See also

Floating-Point Support
exp, expf, expl
log, logf, log10, log10f
sqrt, sqrtf, sqrtl
_CIpow

printf, _printf_l, wprintf, _wprintf_l

10/31/2018 • 3 minutes to read • Edit Online

Prints formatted output to the standard output stream. More secure versions of these functions are available; see printf_s, _printf_s_l, wprintf_s, _wprintf_s_l.

Syntax

```
int printf(
   const char *format [,
   argument]...
);
int _printf_l(
   const char *format,
   locale_t locale [,
   argument]...
);
int wprintf(
   const wchar_t *format [,
   argument]...
);
int _wprintf_l(
   const wchar_t *format,
   locale_t locale [,
   argument]...
);
```

Parameters

format
Format control.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of characters printed, or a negative value if an error occurs. If *format* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**. If **EOF** (0xFFFF) is encountered in *argument*, the function returns -1.

For information on **errno** and error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **printf** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments. **printf** and fprintf behave identically except that **printf** writes output to **stdout** rather than to a destination of type **FILE**.

wprintf is a wide-character version of **printf**; *format* is a wide-character string. **wprintf** and **printf**

behave identically if the stream is opened in ANSI mode. **printf** does not currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tprintf	**printf**	**printf**	**wprintf**

The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are copied to **stdout** in order of their appearance. For example, the line:

```
printf("Line one\n\t\tLine two\n");
```

produces the output:

```
Line one
        Line two
```

Format specifications always begin with a percent sign (**%**) and are read left to right. When **printf** encounters the first format specification (if any), it converts the value of the first argument after *format* and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tprintf	**printf**	**printf**	**wprintf**
_tprintf_l	**_printf_l**	**_printf_l**	**_wprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
printf, **_printf_l**	<stdio.h>
wprintf, **_wprintf_l**	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream

handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_printf.c
// This program uses the printf and wprintf functions
// to produce formatted output.

#include <stdio.h>

int main( void )
{
   char     ch = 'h',
            *string = "computer";
   wchar_t  wch = L'w',
            *wstring = L"Unicode";
   int      count = -9234;
   double   fp = 251.7366;

   // Display integers
   printf( "Integer formats:\n"
           "   Decimal: %d  Justified: %.6d  "
           "Unsigned: %u\n",
           count, count, count, count );

   // Display decimals
   printf( "Decimal %d as:\n   Hex: %Xh  "
           "C hex: 0x%x  Octal: %o\n",
            count, count, count, count );

   // Display in different radixes
   printf( "Digits 10 equal:\n   Hex: %i  "
           "Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );

   // Display characters
   printf("Characters in field (1):\n"
          "%10c%5hc%5C%5lc\n",
          ch, ch, wch, wch);
   wprintf(L"Characters in field (2):\n"
           L"%10C%5hc%5c%5lc\n",
           ch, ch, wch, wch);

   // Display strings
   printf("Strings in field (1):\n%25s\n"
          "%25.4hs\n   %S%25.3ls\n",
          string, string, wstring, wstring);
   wprintf(L"Strings in field (2):\n%25S\n"
           L"%25.4hs\n   %s%25.3ls\n",
           string, string, wstring, wstring);

   // Display real numbers
   printf("Real numbers:\n   %f %.2f %e %E\n",
          fp, fp, fp, fp );

   // Display pointer
   printf( "\nAddress as:   %p\n", &count);
}
```

Sample Output

```
Integer formats:
   Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal -9234 as:
   Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Digits 10 equal:
   Hex: 16  Octal: 8  Decimal: 10
Characters in field (1):
         h    h    w    w
Characters in field (2):
         h    h    w    w
Strings in field (1):
                 computer
                     comp
   Unicode                       Uni
Strings in field (2):
                 computer
                     comp
   Unicode                       Uni
Real numbers:
   251.736600 251.74 2.517366e+002 2.517366E+002

Address as:   0012FF3C
```

See also

Floating-Point Support
Stream I/O
Locale
fopen, _wfopen
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vprintf Functions
_set_output_format

_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Prints formatted output to the standard output stream, and enables specification of the order in which parameters are used in the format string.

Syntax

```
int _printf_p(
   const char *format [,
   argument]...
);
int _printf_p_l(
   const char *format,
   locale_t locale [,
   argument]...
);
int _wprintf_p(
   const wchar_t *format [,
   argument]...
);
int _wprintf_p_l(
   const wchar_t *format,
   locale_t locale [,
   argument]...
);
```

Parameters

format
Format control.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of characters printed or a negative value if an error occurs.

Remarks

The **_printf_p** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments (see printf_p Positional Parameters).

The difference between **_printf_p** and **printf_s** is that **_printf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

_wprintf_p is the wide-character version of **_printf_p**; they behave identically if the stream is opened in ANSI mode. **_printf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
>
> Ensure that *format* is not a user-defined string.

If *format* or *argument* are **NULL**, or of the format string contains invalid formatting characters, **_printf_p** and **_wprintf_p** functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tprintf_p	**_printf_p**	**_printf_p**	**_wprintf_p**
_tprintf_p_l	**_printf_p_l**	**_printf_p_l**	**_wprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_printf_p, **_printf_p_l**	<stdio.h>
_wprintf_p, **_wprintf_p_l**	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_printf_p.c
// This program uses the _printf_p and _wprintf_p
// functions to choose the order in which parameters
// are used.

#include <stdio.h>

int main( void )
{
   // Positional arguments
   _printf_p( "Specifying the order: %2$s %3$s %1$s %4$s %5$s.\n",
              "little", "I'm", "a", "tea", "pot");

   // Resume arguments
   _wprintf_p( L"Reusing arguments: %1$d %1$d %1$d %1$d\n", 10);

   // Width argument
   _printf_p("Width specifiers: %1$*2$s", "Hello\n", 10);
}
```

```
Specifying the order: I'm a little tea pot.
Reusing arguments: 10 10 10 10
Width specifiers:     Hello
```

See also

Floating-Point Support
Stream I/O
Locale
fopen, _wfopen
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l
scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
vprintf Functions

printf_s, _printf_s_l, wprintf_s, _wprintf_s_l

11/9/2018 • 3 minutes to read • Edit Online

Prints formatted output to the standard output stream. These versions of printf, _printf_l, wprintf, _wprintf_l have security enhancements, as described in Security Features in the CRT.

Syntax

```
int printf_s(
   const char *format [,
   argument]...
);
int _printf_s_l(
   const char *format,
   locale_t locale [,
   argument]...
);
int wprintf_s(
   const wchar_t *format [,
   argument]...
);
int _wprintf_s_l(
   const wchar_t *format,
   locale_t locale [,
   argument]...
);
```

Parameters

format
Format control.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of characters printed, or a negative value if an error occurs.

Remarks

The **printf_s** function formats and prints a series of characters and values to the standard output stream, **stdout**. If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments.

The main difference between **printf_s** and **printf** is that **printf_s** checks the format string for valid formatting characters, whereas **printf** only checks if the format string is a null pointer. If either check fails, an invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

For information on **errno** and error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

printf_s and **fprintf_s** behave identically except that **printf_s** writes output to **stdout** rather than to a destination of type **FILE**. For more information, see fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l.

wprintf_s is a wide-character version of **printf_s**; *format* is a wide-character string. **wprintf_s** and **printf_s** behave identically if the stream is opened in ANSI mode. **printf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tprintf_s	**printf_s**	**printf_s**	**wprintf_s**
_tprintf_s_l	**_printf_s_l**	**_printf_s_l**	**_wprintf_s_l**

The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are copied to **stdout** in order of their appearance. For example, the line

```
printf_s("Line one\n\t\tLine two\n");
```

produces the output

```
Line one
        Line two
```

Format specifications always begin with a percent sign (**%**) and are read left to right. When **printf_s** encounters the first format specification (if any), it converts the value of the first argument after *format* and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Requirements

ROUTINE	REQUIRED HEADER
printf_s, **_printf_s_l**	<stdio.h>
wprintf_s, **_wprintf_s_l**	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_printf_s.c
/* This program uses the printf_s and wprintf_s functions
* to produce formatted output.
*/

#include <stdio.h>

int main( void )
{
   char   ch = 'h', *string = "computer";
   int    count = -9234;
   double fp = 251.7366;
   wchar_t wch = L'w', *wstring = L"Unicode";

   /* Display integers. */
   printf_s( "Integer formats:\n"
           "   Decimal: %d  Justified: %.6d  Unsigned: %u\n",
           count, count, count );

   printf_s( "Decimal %d as:\n   Hex: %Xh  C hex: 0x%x  Octal: %o\n",
            count, count, count, count );

   /* Display in different radixes. */
   printf_s( "Digits 10 equal:\n   Hex: %i  Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );

   /* Display characters. */

   printf_s("Characters in field (1):\n%10c%5hc%5C%5lc\n", ch, ch, wch, wch);
   wprintf_s(L"Characters in field (2):\n%10C%5hc%5c%5lc\n", ch, ch, wch, wch);

   /* Display strings. */

   printf_s("Strings in field (1):\n%25s\n%25.4hs\n   %S%25.3ls\n",
   string, string, wstring, wstring);
   wprintf_s(L"Strings in field (2):\n%25S\n%25.4hs\n   %s%25.3ls\n",
       string, string, wstring, wstring);

   /* Display real numbers. */
   printf_s( "Real numbers:\n   %f %.2f %e %E\n", fp, fp, fp, fp );

   /* Display pointer. */
   printf_s( "\nAddress as:   %p\n", &count);

}
```

Sample Output

```
Integer formats:
   Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal -9234 as:
   Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Digits 10 equal:
   Hex: 16  Octal: 8  Decimal: 10
Characters in field (1):
         h    h    w    w
Characters in field (2):
         h    h    w    w
Strings in field (1):
                 computer
                     comp
   Unicode                       Uni
Strings in field (2):
                 computer
                     comp
   Unicode                       Uni
Real numbers:
   251.736600 251.74 2.517366e+002 2.517366E+002

Address as:   0012FF78
```

See also

Floating-Point Support
Stream I/O
Locale
fopen, _wfopen
fprintf, _fprintf_l, fwprintf, _fwprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vprintf Functions

_purecall

5/8/2019 • 2 minutes to read • Edit Online

The default pure virtual function call error handler. The compiler generates code to call this function when a pure virtual member function is called.

Syntax

```
extern "C" int __cdecl _purecall();
```

Remarks

The **_purecall** function is a Microsoft-specific implementation detail of the Microsoft C++ compiler. This function is not intended to be called by your code directly, and it has no public header declaration. It is documented here because it is a public export of the C Runtime Library.

A call to a pure virtual function is an error because it has no implementation. The compiler generates code to invoke the **_purecall** error handler function when a pure virtual function is called. By default, **_purecall** terminates the program. Before terminating, the **_purecall** function invokes a **_purecall_handler** function if one has been set for the process. You can install your own error handler function for pure virtual function calls, to catch them for debugging or reporting purposes. To use your own error handler, create a function that has the **_purecall_handler** signature, then use _set_purecall_handler to make it the current handler.

Requirements

The **_purecall** function does not have a header declaration. The **_purecall_handler** typedef is defined in <stdlib.h>.

See also

Alphabetical Function Reference
_get_purecall_handler, _set_purecall_handler

putc, putwc

11/8/2018 • 2 minutes to read • Edit Online

Writes a character to a stream.

Syntax

```
int putc(
   int c,
   FILE *stream
);
wint_t putwc(
   wchar_t c,
   FILE *stream
);
```

Parameters

c
Character to be written.

stream
Pointer to **FILE** structure.

Return Value

Returns the character written. To indicate an error or end-of-file condition, **putc** and **putchar** return **EOF**; **putwc** and **putwchar** return **WEOF**. For all four routines, use ferror or feof to check for an error or end of file. If passed a null pointer for *stream*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** or **WEOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **putc** routine writes the single character *c* to the output *stream* at the current position. Any integer can be passed to **putc**, but only the lower 8 bits are written. The **putchar** routine is identical to `putc( c, stdout )`. For each routine, if a read error occurs, the error indicator for the stream is set. **putc** and **putchar** are similar to **fputc** and **_fputchar**, respectively, but are implemented both as functions and as macros (see Choosing Between Functions and Macros). **putwc** and **putwchar** are wide-character versions of **putc** and **putchar**, respectively. **putwc** and **putc** behave identically if the stream is opened in ANSI mode. **putc** doesn't currently support output into a UNICODE stream.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see **_putc_nolock, _putwc_nolock**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttc	**putc**	**putc**	**putwc**

Requirements

ROUTINE	REQUIRED HEADER
putc	<stdio.h>
putwc	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putc.c
/* This program uses putc to write buffer
* to a stream. If an error occurs, the program
* stops before writing the entire buffer.
*/

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putc( *p, stream );
}
```

Output

```
This is the line of output
```

See also

Stream I/O
fputc, fputwc
getc, getwc

_putc_nolock, _putwc_nolock

10/31/2018 • 2 minutes to read • Edit Online

Writes a character to a stream without locking the thread.

Syntax

```
int _putc_nolock(
   int c,
   FILE *stream
);
wint_t _putwc_nolock(
   wchar_t c,
   FILE *stream
);
```

Parameters

c
Character to be written.

stream
Pointer to the **FILE** structure.

Return Value

See **putc, putwc**.

Remarks

_putc_nolock and **_putwc_nolock** are identical to the versions without the **_nolock** suffix except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

_putwc_nolock is the wide-character version of **_putc_nolock**; the two functions behave identically if the stream is opened in ANSI mode. **_putc_nolock** doesn't currently support output into a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttc_nolock	**_putc_nolock**	**_putc_nolock**	**_putwc_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_putc_nolock	<stdio.h>
_putwc_nolock	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putc_nolock.c
/* This program uses putc to write buffer
* to a stream. If an error occurs, the program
* stops before writing the entire buffer.
*/

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = _putc_nolock( *p, stream );
}
```

Output

```
This is the line of output
```

See also

Stream I/O
fputc, fputwc
getc, getwc

putch

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _putch instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_putch, _putwch

11/8/2018 • 2 minutes to read • Edit Online

Writes a character to the console.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _putch(
   int c
);

wint_t _putwch(
   wchar_t c
);
```

Parameters

c
Character to be output.

Return Value

Returns *c* if successful. If **_putch** fails, it returns **EOF**; if **_putwch** fails, it returns **WEOF**.

Remarks

These functions write the character *c* directly, without buffering, to the console. In Windows NT, **_putwch** writes Unicode characters using the current console locale setting.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. For more information, see **_putch_nolock**, **_putwch_nolock**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttch	**_putch**	**_putch**	**_putwch**

Requirements

ROUTINE	REQUIRED HEADER
_putch	<conio.h>
_putwch	<conio.h>

ROUTINE	REQUIRED HEADER

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

See the example for _getch.

See also

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
_getch, _getwch

_putch_nolock, _putwch_nolock

10/31/2018 • 2 minutes to read • Edit Online

Writes a character to the console without locking the thread.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _putch_nolock(
int c
);
wint_t _putwch_nolock(
wchar_t c
);
```

Parameters

c
Character to be output.

Return Value

Returns *c* if successful. If **_putch_nolock** fails, it returns **EOF**; if **_putwch_nolock** fails, it returns **WEOF**.

Remarks

_putch_nolock and **_putwch_nolock** are identical to **_putch** and **_putwch**, respectively, except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttch_nolock	**_putch_nolock**	**_putch_nolock**	**_putwch_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_putch_nolock	<conio.h>
_putwch_nolock	<conio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
_getch, _getwch

putchar, putwchar

11/8/2018 • 2 minutes to read • Edit Online

Writes a character to **stdout**.

Syntax

```
int putchar(
   int c
);
wint_t putwchar(
   wchar_t c
);
```

Parameters

c
Character to be written.

Return Value

Returns the character written. To indicate an error or end-of-file condition, **putc** and **putchar** return **EOF**; **putwc** and **putwchar** return **WEOF**. For all four routines, use ferror or feof to check for an error or end of file. If passed a null pointer for *stream*, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, they return **EOF** or **WEOF** and set **errno** to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, error codes.

Remarks

The **putc** routine writes the single character *c* to the output *stream* at the current position. Any integer can be passed to **putc**, but only the lower 8 bits are written. The **putchar** routine is identical to `putc( c, stdout )`. For each routine, if a read error occurs, the error indicator for the stream is set. **putc** and **putchar** are similar to **fputc** and **_fputchar**, respectively, but are implemented both as functions and as macros (see Choosing Between Functions and Macros). **putwc** and **putwchar** are wide-character versions of **putc** and **putchar**, respectively.

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttchar	**putchar**	**putchar**	**putwchar**

Requirements

ROUTINE	REQUIRED HEADER
putchar	<stdio.h>
putwchar	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putchar.c
/* This program uses putc to write buffer
* to a stream. If an error occurs, the program
* stops before writing the entire buffer.
*/

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;

   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putchar( *p );
}
```

Output

```
This is the line of output
```

See also

Stream I/O
fputc, fputwc
getc, getwc

_putchar_nolock, _putwchar_nolock

11/9/2018 • 2 minutes to read • Edit Online

Writes a character to **stdout** without locking the thread.

Syntax

```
int _putchar_nolock(
   int c
);
wint_t _putwchar_nolock(
   wchar_t c
);
```

Parameters

c
Character to be written.

Return Value

See **putchar, putwchar**.

Remarks

putchar_nolock and **_putwchar_nolock** are identical to the versions without the **_nolock** suffix except that they are not protected from interference by other threads. They might be faster because they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_puttchar_nolock	**_putchar_nolock**	**_putchar_nolock**	**_putwchar_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_putchar_nolock	<stdio.h>
_putwchar_nolock	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putchar_nolock.c
/* This program uses putchar to write buffer
* to stdout. If an error occurs, the program
* stops before writing the entire buffer.
*/

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;

   ch = 0;

   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = _putchar_nolock( *p );
}
```

Output

```
This is the line of output
```

See also

Stream I/O
fputc, fputwc
fgetc, fgetwc

putenv

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _putenv or security-enhanced _putenv_s instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_putenv, _wputenv

10/31/2018 • 2 minutes to read • Edit Online

Creates, modifies, or removes environment variables. More secure versions of these functions are available; see _putenv_s, _wputenv_s.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _putenv(
   const char *envstring
);
int _wputenv(
   const wchar_t *envstring
);
```

Parameters

envstring
Environment-string definition.

Return Value

Return 0 if successful or -1 in the case of an error.

Remarks

The **_putenv** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program). **_wputenv** is a wide-character version of **_putenv**; the *envstring* argument to **_wputenv** is a wide-character string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tputenv	**_putenv**	**_putenv**	**_wputenv**

The *envstring* argument must be a pointer to a string of the form *varname*=*value_string*, where *varname* is the name of the environment variable to be added or modified and *value_string* is the variable's value. If *varname* is already part of the environment, its value is replaced by *value_string*; otherwise, the new *varname* variable and its *value_string* value are added to the environment. You can remove a variable from the environment by specifying an empty *value_string*, or in other words, by specifying only *varname*=.

_putenv and **_wputenv** affect only the environment that is local to the current process; you cannot use them to modify the command-level environment. That is, these functions operate only on data structures accessible

to the run-time library and not on the environment segment created for a process by the operating system. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level). However, the modified environment can be passed to any new processes created by **_spawn**, **_exec**, or **system**, and these new processes get any new items added by **_putenv** and **_wputenv**.

Do not change an environment entry directly: instead, use **_putenv** or **_wputenv** to change it. In particular, direct freeing elements of the **_environ[]** global array might lead to invalid memory being addressed.

getenv and **_putenv** use the global variable **_environ** to access the environment table; **_wgetenv** and **_wputenv** use **_wenviron**. **_putenv** and **_wputenv** might change the value of **_environ** and **_wenviron**, thus invalidating the **_envp** argument to **main** and the **_wenvp** argument to **wmain**. Therefore, it is safer to use **_environ** or **_wenviron** to access the environment information. For more information about the relation of **_putenv** and **_wputenv** to global variables, see _environ, _wenviron.

> **NOTE**
> The **_putenv** and **_getenv** families of functions are not thread-safe. **_getenv** could return a string pointer while **_putenv** is modifying the string, causing random failures. Make sure that calls to these functions are synchronized.

Requirements

ROUTINE	REQUIRED HEADER
_putenv	<stdlib.h>
_wputenv	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

For a sample of how to use **_putenv**, see getenv, _wgetenv.

See also

Process and Environment Control
getenv, _wgetenv
_searchenv, _wsearchenv

_putenv_s, _wputenv_s

10/31/2018 • 2 minutes to read • Edit Online

Creates, modifies, or removes environment variables. These are versions of _putenv, _wputenv but have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _putenv_s(
   const char *varname,
   const char *value_string
);
errno_t _wputenv_s(
   const wchar_t *varname,
   const wchar_t *value_string
);
```

Parameters

varname
The environment variable name.

value_string
The value to set the environment variable to.

Return Value

Returns 0 if successful, or an error code.

Error Conditions

VARNAME	*VALUE_STRING*	RETURN VALUE
NULL	any	**EINVAL**
any	**NULL**	**EINVAL**

If one of the error conditions occurs, these functions invoke an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Remarks

The **_putenv_s** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program). **_wputenv_s** is a wide-character version of **_putenv_s**; the *envstring* argument to **_wputenv_s** is a wide-character string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tputenv_s	**_putenv_s**	**_putenv_s**	**_wputenv_s**

varname is the name of the environment variable to be added or modified and *value_string* is the variable's value. If *varname* is already part of the environment, its value is replaced by *value_string*; otherwise, the new *varname* variable and its *value_string* are added to the environment. You can remove a variable from the environment by specifying an empty string (that is, "") for *value_string*.

_putenv_s and **_wputenv_s** affect only the environment that is local to the current process; you cannot use them to modify the command-level environment. These functions operate only on data structures that are accessible to the run-time library and not on the environment "segment" that the operating system creates for a process. When the current process terminates, the environment reverts to the level of the calling process, which in most cases is the operating-system level. However, the modified environment can be passed to any new processes that are created by **_spawn**, **_exec**, or **system**, and these new processes get any new items that are added by **_putenv_s** and **_wputenv_s**.

Do not change an environment entry directly; instead, use **_putenv_s** or **_wputenv_s** to change it. In particular, directly freeing elements of the **_environ[]** global array might cause invalid memory to be addressed.

getenv and **_putenv_s** use the global variable **_environ** to access the environment table; **_wgetenv** and **_wputenv_s** use **_wenviron**. **_putenv_s** and **_wputenv_s** may change the value of **_environ** and **_wenviron**, and thereby invalidate the *envp* argument to **main** and the **_wenvp** argument to **wmain**. Therefore, it is safer to use **_environ** or **_wenviron** to access the environment information. For more information about the relationship of **_putenv_s** and **_wputenv_s** to global variables, see _environ, _wenviron.

> **NOTE**
> The **_putenv_s** and **_getenv_s** families of functions are not thread-safe. **_getenv_s** could return a string pointer while **_putenv_s** is modifying the string, and thereby cause random failures. Make sure that calls to these functions are synchronized.

Requirements

ROUTINE	REQUIRED HEADER
_putenv_s	<stdlib.h>
_wputenv_s	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

For a sample that shows how to use **_putenv_s**, see getenv_s, _wgetenv_s.

See also

Process and Environment Control
getenv, _wgetenv

puts, _putws

10/31/2018 • 2 minutes to read • Edit Online

Writes a string to **stdout**.

Syntax

```
int puts(
   const char *str
);
int _putws(
   const wchar_t *str
);
```

Parameters

str
Output string.

Return Value

Returns a nonnegative value if successful. If **puts** fails, it returns **EOF**; if **_putws** fails, it returns **WEOF**. If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions set **errno** to **EINVAL** and return **EOF** or **WEOF**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **puts** function writes *str* to the standard output stream **stdout**, replacing the string's terminating null character ('\0') with a newline character ('\n') in the output stream.

_putws is the wide-character version of **puts**; the two functions behave identically if the stream is opened in ANSI mode. **puts** doesn't currently support output into a UNICODE stream.

_putwch writes Unicode characters using the current CONSOLE LOCALE setting.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_putts	**puts**	**puts**	**_putws**

Requirements

ROUTINE	REQUIRED HEADER
puts	<stdio.h>
_putws	<stdio.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_puts.c
// This program uses puts to write a string to stdout.

#include <stdio.h>

int main( void )
{
   puts( "Hello world from puts!" );
}
```

Output

```
Hello world from puts!
```

See also

Stream I/O
fputs, fputws
fgets, fgetws

putw

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _putw instead.

_putw

10/31/2018 • 2 minutes to read • Edit Online

Writes an integer to a stream.

Syntax

```
int _putw(
   int binint,
   FILE *stream
);
```

Parameters

binint
Binary integer to be output.

stream
Pointer to the **FILE** structure.

Return Value

Returns the value written. A return value of **EOF** might indicate an error. Because **EOF** is also a legitimate integer value, use **ferror** to verify an error. If *stream* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EOF**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_putw** function writes a binary value of type **int** to the current position of *stream*. **_putw** does not affect the alignment of items in the stream nor does it assume any special alignment. **_putw** is primarily for compatibility with previous libraries. Portability problems might occur with **_putw** because the size of an **int** and the ordering of bytes within an **int** differ across systems.

Requirements

ROUTINE	REQUIRED HEADER
_putw	<stdio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_putw.c
/* This program uses _putw to write a
* word to a stream, then performs an error check.
*/

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   FILE *stream;
   unsigned u;
   if( fopen_s( &stream, "data.out", "wb" ) )
      exit( 1 );
   for( u = 0; u < 10; u++ )
   {
      _putw( u + 0x2132, stream );   /* Write word to stream. */
      if( ferror( stream ) )         /* Make error check. */
      {
         printf( "_putw failed" );
         clearerr_s( stream );
         exit( 1 );
      }
   }
   printf( "Wrote ten words\n" );
   fclose( stream );
}
```

Output

```
Wrote ten words
```

See also

Stream I/O
_getw

_query_new_handler

10/31/2018 • 2 minutes to read • Edit Online

Returns the address of the current new handler routine.

Syntax

```
_PNH _query_new_handler(
   void
);
```

Return Value

Returns the address of the current new handler routine as set by **_set_new_handler**.

Remarks

The C++ **_query_new_handler** function returns the address of the current exception-handling function set by the C++ _set_new_handler function. **_set_new_handler** is used to specify an exception-handling function that is to gain control if the **new** operator fails to allocate memory. For more information, see the discussion of the new and delete operators in the C++ Language Reference.

Requirements

ROUTINE	REQUIRED HEADER
_query_new_handler	<new.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Memory Allocation
free

_query_new_mode

10/31/2018 • 2 minutes to read • Edit Online

Returns an integer indicating the new handler mode set by **_set_new_mode** for **malloc**.

Syntax

```
int _query_new_mode(
   void
);
```

Return Value

Returns the current new handler mode, namely 0 or 1, for **malloc**. A return value of 1 indicates that, on failure to allocate memory, **malloc** calls the new handler routine; a return value of 0 indicates that it does not.

Remarks

The C++ **_query_new_mode** function returns an integer that indicates the new handler mode that is set by the C++ _set_new_mode function for malloc. The new handler mode indicates whether, on failure to allocate memory, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure. You can use **_set_new_mode** to override this behavior so that on failure **malloc** calls the new handler routine in the same way that the **new** operator does when it fails to allocate memory. For more information, see the discussion of the new and delete operators in the C++ Language Reference.

Requirements

ROUTINE	REQUIRED HEADER
_query_new_mode	<new.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

Memory Allocation
calloc
free
realloc
_query_new_handler

quick_exit

10/31/2018 • 2 minutes to read • Edit Online

Causes normal program termination to occur.

Syntax

```
__declspec(noreturn) void quick_exit(
    int status
);
```

Parameters

status
The status code to return to the host environment.

Return Value

The **quick_exit** function cannot return to its caller.

Remarks

The **quick_exit** function causes normal program termination. It calls no functions registered by **atexit**, **_onexit** or signal handlers registered by the **signal** function. Behavior is undefined if **quick_exit** is called more than once, or if the **exit** function is also called.

The **quick_exit** function calls, in last-in, first-out (LIFO) order, the functions registered by **at_quick_exit**, except for those functions already called when the function was registered. Behavior is undefined if a longjmp call is made during a call to a registered function that would terminate the call to the function.

After the registered functions have been called, **quick_exit** invokes **_Exit** by using the *status* value to return control to the host environment.

Requirements

ROUTINE	REQUIRED HEADER
quick_exit	<process.h> or <stdlib.h>

For more information about compatibility, see Compatibility.

See also

Process and Environment Control
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_onexit, _onexit_m
_spawn, _wspawn Functions

system, _wsystem

qsort

3/1/2019 • 2 minutes to read • Edit Online

Performs a quick sort. A more secure version of this function is available; see qsort_s.

Syntax

```
void qsort(
   void *base,
   size_t number,
   size_t width,
   int (__cdecl *compare )(const void *, const void *)
);
```

Parameters

base
Start of target array.

number
Array size in elements.

width
Element size in bytes.

compare
Pointer to a user-supplied routine that compares two array elements and returns a value that specifies their relationship.

Remarks

The **qsort** function implements a quick-sort algorithm to sort an array of *number* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. **qsort** overwrites this array by using the sorted elements.

qsort calls the *compare* routine one or more times during the sort, and passes pointers to two array elements on each call.

```
compare( (void *) & elem1, (void *) & elem2 );
```

The routine compares the elements and returns one of the following values.

COMPARE FUNCTION RETURN VALUE	DESCRIPTION
< 0	**elem1** less than **elem2**
0	**elem1** equivalent to **elem2**
> 0	**elem1** greater than **elem2**

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

This function validates its parameters. If *compare* or *number* is **NULL**, or if *base* is **NULL** and *number* is nonzero, or if *width* is less than zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns and **errno** is set to **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
qsort	<stdlib.h> and <search.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_qsort.c
// arguments: every good boy deserves favor

/* This program reads the command-line
* parameters and uses qsort to sort them. It
* then displays the sorted arguments.
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int compare( const void *arg1, const void *arg2 );

int main( int argc, char **argv )
{
   int i;
   /* Eliminate argv[0] from sort: */
   argv++;
   argc--;

   /* Sort remaining args using Quicksort algorithm: */
   qsort( (void *)argv, (size_t)argc, sizeof( char * ), compare );

   /* Output sorted list: */
   for( i = 0; i < argc; ++i )
      printf( " %s", argv[i] );
   printf( "\n" );
}

int compare( const void *arg1, const void *arg2 )
{
   /* Compare all of both strings: */
   return _stricmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```

```
boy deserves every favor good
```

See also

Searching and Sorting
bsearch
_lsearch

qsort_s

3/1/2019 • 4 minutes to read • Edit Online

Performs a quick sort. A version of qsort with security enhancements as described in Security Features in the CRT.

Syntax

```
void qsort_s(
   void *base,
   size_t num,
   size_t width,
   int (__cdecl *compare )(void *, const void *, const void *),
   void * context
);
```

Parameters

base
Start of target array.

number
Array size in elements.

width
Element size in bytes.

compare
Comparison function. The first argument is the *context* pointer. The second argument is a pointer to the *key* for the search. The third argument is a pointer to the array element to be compared with *key*.

context
A pointer to a context, which can be any object that the *compare* routine needs to access.

Remarks

The **qsort_s** function implements a quick-sort algorithm to sort an array of *number* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. **qsort_s** overwrites this array with the sorted elements. The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort_s** calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

```
compare( context, (void *) & elem1, (void *) & elem2 );
```

The routine must compare the elements and then return one of the following values:

RETURN VALUE	DESCRIPTION
< 0	**elem1** less than **elem2**
0	**elem1** equivalent to **elem2**

RETURN VALUE	DESCRIPTION
> 0	**elem1** greater than **elem2**

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

If invalid parameters are passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, then the function returns and **errno** is set to **EINVAL**. For more information, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Error Conditions

KEY	BASE	COMPARE	NUM	WIDTH	ERRNO
NULL	any	any	any	any	**EINVAL**
any	**NULL**	any	!= 0	any	**EINVAL**
any	any	any	any	<= 0	**EINVAL**
any	any	**NULL**	any	any	**EINVAL**

qsort_s has the same behavior as **qsort** but has the *context* parameter and sets **errno**. By passing a *context* parameter, comparison functions can use an object pointer to access object functionality or other information not accessible through an element pointer. The addition of the *context* parameter makes **qsort_s** more secure because *context* can be used to avoid reentrancy bugs introduced by using static variables to make shared information available to the *compare* function.

Requirements

ROUTINE	REQUIRED HEADER
qsort_s	<stdlib.h> and <search.h>

For additional compatibility information, see Compatibility.

Libraries: All versions of the CRT Library Features.

Example

The following example demonstrates how to use the *context* parameter in the **qsort_s** function. The *context* parameter makes it easier to perform thread-safe sorts. Instead of using static variables that must be synchronized to ensure thread safety, pass a different *context* parameter in each sort. In this example, a locale object is used as the *context* parameter.

```
// crt_qsort_s.cpp
// compile with: /EHsc /MT
#include <stdlib.h>
#include <stdio.h>
#include <search.h>
#include <process.h>
#include <locale.h>
#include <locale>
#include <windows.h>
using namespace std;
```

```
// The sort order is dependent on the code page.  Use 'chcp' at the
// command line to change the codepage.  When executing this application,
// the command prompt codepage must match the codepage used here:

#define CODEPAGE_850

#ifdef CODEPAGE_850
// Codepage 850 is the OEM codepage used by the command line,
// so \x00e1 is the German Sharp S in that codepage and \x00a4
// is the n tilde.

char *array1[] = { "wei\x00e1", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };
char *array2[] = { "Espa\x00a4ol", "Espa\x00a4" "a", "espantado" };
char *array3[] = { "table", "tableux", "tablet" };

#define GERMAN_LOCALE "German_Germany.850"
#define SPANISH_LOCALE "Spanish_Spain.850"
#define ENGLISH_LOCALE "English_US.850"

#endif

#ifdef CODEPAGE_1252
   // If using codepage 1252 (ISO 8859-1, Latin-1), use \x00df
   // for the German Sharp S and \x001f for the n tilde.
char *array1[] = { "wei\x00df", "weis", "annehmen", "weizen", "Zeit",
                   "weit" };
char *array2[] = { "Espa\x00f1ol", "Espa\x00f1" "a", "espantado" };
char *array3[] = { "table", "tableux", "tablet" };

#define GERMAN_LOCALE "German_Germany.1252"
#define SPANISH_LOCALE "Spanish_Spain.1252"
#define ENGLISH_LOCALE "English_US.1252"

#endif

// The context parameter lets you create a more generic compare.
// Without this parameter, you would have stored the locale in a
// static variable, thus making sort_array vulnerable to thread
// conflicts.

int compare( void *pvlocale, const void *str1, const void *str2)
{
    char s1[256];
    char s2[256];
    strcpy_s(s1, 256, *(char**)str1);
    strcpy_s(s2, 256, *(char**)str2);
    _strlwr_s( s1, sizeof(s1) );
    _strlwr_s( s2, sizeof(s2) );

    locale& loc = *( reinterpret_cast< locale * > ( pvlocale));

    return use_facet< collate<char> >(loc).compare(s1,
       &s1[strlen(s1)], s2, &s2[strlen(s2)]);

}

void sort_array(char *array[], int num, locale &loc)
{
    qsort_s(array, num, sizeof(char*), compare, &loc);
}

void print_array(char *a[], int c)
{
   for (int i = 0; i < c; i++)
      printf("%s ", a[i]);
   printf("\n");
```

```
}

void sort_german(void * Dummy)
{
   sort_array(array1, 6, locale(GERMAN_LOCALE));
}

void sort_spanish(void * Dummy)
{
   sort_array(array2, 3, locale(SPANISH_LOCALE));
}

void sort_english(void * Dummy)
{
   sort_array(array3, 3, locale(ENGLISH_LOCALE));
}

int main( )
{
   int i;
   HANDLE threads[3];

   printf("Unsorted input:\n");
   print_array(array1, 6);
   print_array(array2, 3);
   print_array(array3, 3);

   // Create several threads that perform sorts in different
   // languages at the same time.

   threads[0] = reinterpret_cast<HANDLE>(
                _beginthread( sort_german , 0, NULL));
   threads[1] = reinterpret_cast<HANDLE>(
                _beginthread( sort_spanish, 0, NULL));
   threads[2] = reinterpret_cast<HANDLE>(
                _beginthread( sort_english, 0, NULL));

   for (i = 0; i < 3; i++)
   {
      if (threads[i] == reinterpret_cast<HANDLE>(-1))
      {
         printf("Error creating threads.\n");
         exit(1);
      }
   }

   // Wait until all threads have terminated.
   WaitForMultipleObjects(3, threads, true, INFINITE);

   printf("Sorted output: \n");

   print_array(array1, 6);
   print_array(array2, 3);
   print_array(array3, 3);
}
```

Sample Output

```
Unsorted input:
weiß weis annehmen weizen Zeit weit
Español España espantado
table tableux tablet
Sorted output:
annehmen weiß weis weit weizen Zeit
España Español espantado
table tablet tableux
```

See also

Searching and Sorting
bsearch_s
_lsearch_s
qsort

raise

10/31/2018 • 2 minutes to read • Edit Online

Sends a signal to the executing program.

> **NOTE**
> Do not use this method to shut down a Microsoft Store app, except in testing or debugging scenarios. Programmatic or UI ways to close a Store app are not permitted according to the Microsoft Store policies. For more information, see UWP app lifecycle.

Syntax

```
int raise(
   int sig
);
```

Parameters

sig
Signal to be raised.

Return Value

If successful, **raise** returns 0. Otherwise, it returns a nonzero value.

Remarks

The **raise** function sends *sig* to the executing program. If a previous call to **signal** has installed a signal-handling function for *sig*, **raise** executes that function. If no handler function has been installed, the default action associated with the signal value *sig* is taken, as follows.

SIGNAL	MEANING	DEFAULT
SIGABRT	Abnormal termination	Terminates the calling program with exit code 3
SIGFPE	Floating-point error	Terminates the calling program
SIGILL	Illegal instruction	Terminates the calling program
SIGINT	CTRL+C interrupt	Terminates the calling program
SIGSEGV	Illegal storage access	Terminates the calling program
SIGTERM	Termination request sent to the program	Ignores the signal

If the argument is not a valid signal as specified above, the invalid parameter handler is invoked, as described in Parameter Validation. If not handled, the function sets **errno** to **EINVAL** and returns a nonzero value.

Requirements

ROUTINE	REQUIRED HEADER
raise	<signal.h>

For additional compatibility information, see Compatibility.

See also

Process and Environment Control
abort
signal

rand

3/1/2019 • 2 minutes to read • Edit Online

Generates a pseudorandom number by using a well-known and fully-reproducible algorithm. A more programmatically secure version of this function is available; see rand_s. Numbers generated by **rand** are not cryptographically secure. For more cryptographically secure random number generation, use rand_s or the functions declared in the C++ Standard Library in <random>.

Syntax

```
int rand( void );
```

Return Value

rand returns a pseudorandom number, as described above. There is no error return.

Remarks

The **rand** function returns a pseudorandom integer in the range 0 to **RAND_MAX** (32767). Use the srand function to seed the pseudorandom-number generator before calling **rand**.

The **rand** function generates a well-known sequence and is not appropriate for use as a cryptographic function. For more cryptographically secure random number generation, use rand_s or the functions declared in the C++ Standard Library in <random>. For information about what's wrong with **rand** and how <random> addresses these shortcomings, see this video entitled rand Considered Harmful.

Requirements

ROUTINE	REQUIRED HEADER
rand	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

```c
// crt_rand.c
// This program seeds the random-number generator
// with the time, then exercises the rand function.
//

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void SimpleRandDemo( int n )
{
   // Print n random numbers.
   int i;
   for( i = 0; i < n; i++ )
      printf( "  %6d\n", rand() );
}

void RangedRandDemo( int range_min, int range_max, int n )
{
   // Generate random numbers in the half-closed interval
   // [range_min, range_max). In other words,
   // range_min <= random number < range_max
   int i;
   for ( i = 0; i < n; i++ )
   {
      int u = (double)rand() / (RAND_MAX + 1) * (range_max - range_min)
            + range_min;
      printf( "  %6d\n", u);
   }
}

int main( void )
{
   // Seed the random-number generator with the current time so that
   // the numbers will be different every time we run.
   srand( (unsigned)time( NULL ) );

   SimpleRandDemo( 10 );
   printf("\n");
   RangedRandDemo( -100, 100, 10 );
}
```

```
22036
18330
11651
27464
18093
3284
11785
14686
11447
11285

   74
   48
   27
   65
   96
   64
   -5
  -42
  -55
   66
```

See also

Floating-Point Support
srand
rand_s

rand_s

10/31/2018 • 2 minutes to read • Edit Online

Generates a pseudorandom number. This is a more secure version of the function rand, with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t rand_s(unsigned int* randomValue);
```

Parameters

randomValue
A pointer to an integer to hold the generated value.

Return Value

Zero if successful, otherwise, an error code. If the input pointer *randomValue* is a null pointer, the function invokes an invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**. If the function fails for any other reason, **randomValue* is set to 0.

Remarks

The **rand_s** function writes a pseudorandom integer in the range 0 to **UINT_MAX** to the input pointer. The **rand_s** function uses the operating system to generate cryptographically secure random numbers. It does not use the seed generated by the srand function, nor does it affect the random number sequence used by rand.

The **rand_s** function requires that constant **_CRT_RAND_S** be defined prior to the inclusion statement for the function to be declared, as in the following example:

```
#define _CRT_RAND_S
#include <stdlib.h>
```

rand_s depends on the RtlGenRandom API, which is only available in Windows XP and later.

Requirements

ROUTINE	REQUIRED HEADER
rand_s	<stdlib.h>

For more information, see Compatibility.

Example

```
// crt_rand_s.c
// This program illustrates how to generate random
// integer or floating point numbers in a specified range.

// Remembering to define _CRT_RAND_S prior
// to inclusion statement.
#define _CRT_RAND_S

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>

int main( void )
{
    int             i;
    unsigned int    number;
    double          max = 100.0;
    errno_t         err;

    // Display 10 random integers in the range [ 1,10 ].
    for( i = 0; i < 10;i++ )
    {
        err = rand_s( &number );
        if (err != 0)
        {
            printf_s("The rand_s function failed!\n");
        }
        printf_s( "  %u\n", (unsigned int) ((double)number /
                        ((double) UINT_MAX + 1 ) * 10.0) + 1);
    }

    printf_s("\n");

    // Display 10 random doubles in [0, max).
    for (i = 0; i < 10;i++ )
    {
        err = rand_s( &number );
        if (err != 0)
        {
            printf_s("The rand_s function failed!\n");
        }
        printf_s( "  %g\n", (double) number /
                             ((double) UINT_MAX + 1) * max );
    }
}
```

Sample Output

```
10
4
5
2
8
2
5
6
1
1

32.6617
29.4471
11.5413
6.41924
20.711
60.2878
61.0094
20.1222
80.9192
65.0712
```

See also

Floating-Point Support
rand
srand

read

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _read instead.

_read

2/14/2019 • 2 minutes to read • Edit Online

Reads data from a file.

Syntax

```
int _read(
   int const fd,
   void * const buffer,
   unsigned const buffer_size
);
```

Parameters

fd
File descriptor referring to the open file.

buffer
Storage location for data.

buffer_size
Maximum number of bytes to read.

Return Value

_read returns the number of bytes read, which might be less than *buffer_size* if there are fewer than *buffer_size* bytes left in the file, or if the file was opened in text mode. In text mode, each carriage return-line feed pair `\r\n` is replaced with a single linefeed character `\n`. Only the single linefeed character is counted in the return value. The replacement does not affect the file pointer.

If the function tries to read at end of file, it returns 0. If *fd* is not valid, the file isn't open for reading, or the file is locked, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EBADF**.

If *buffer* is **NULL**, or if *buffer_size* > **INT_MAX**, the invalid parameter handler is invoked. If execution is allowed to continue, the function returns -1 and **errno** is set to **EINVAL**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_read** function reads a maximum of *buffer_size* bytes into *buffer* from the file associated with *fd*. The read operation begins at the current position of the file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

If the file was opened in text mode, the read terminates when **_read** encounters a CTRL+Z character, which is treated as an end-of-file indicator. Use _lseek to clear the end-of-file indicator.

Requirements

ROUTINE	REQUIRED HEADER
_read	<io.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_read.c
/* This program opens a file named crt_read.txt
* and tries to read 60,000 bytes from
* that file using _read. It then displays the
* actual number of bytes read.
*/

#include <fcntl.h>      /* Needed only for _O_RDWR definition */
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <share.h>

char buffer[60000];

int main( void )
{
   int fh, bytesread;
   unsigned int nbytes = 60000;

   /* Open file for input: */
   if ( _sopen_s( &fh, "crt_read.txt", _O_RDONLY, _SH_DENYNO, 0 ))
   {
      perror( "open failed on input file" );
      exit( 1 );
   }

   /* Read in input: */
   if (( bytesread = _read( fh, buffer, nbytes )) <= 0 )
      perror( "Problem reading file" );
   else
      printf( "Read %u bytes from file\n", bytesread );

   _close( fh );
}
```

Input: crt_read.txt

```
Line one.
Line two.
```

Output

```
Read 19 bytes from file
```

See also

Low-Level I/O
_creat, _wcreat
fread
_open, _wopen
_write

realloc

10/31/2018 • 3 minutes to read • Edit Online

Reallocate memory blocks.

Syntax

```
void *realloc(
   void *memblock,
   size_t size
);
```

Parameters

memblock
Pointer to previously allocated memory block.

size
New size in bytes.

Return Value

realloc returns a **void** pointer to the reallocated (and possibly moved) memory block.

If there is not enough available memory to expand the block to the given size, the original block is left unchanged, and **NULL** is returned.

If *size* is zero, then the block pointed to by *memblock* is freed; the return value is **NULL**, and *memblock* is left pointing at a freed block.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **realloc** function changes the size of an allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is **NULL**, **realloc** behaves the same way as **malloc** and allocates a new block of *size* bytes. If *memblock* is not **NULL**, it should be a pointer returned by a previous call to **calloc**, **malloc**, or **realloc**.

The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block can be in a different location. Because the new block can be in a new memory location, the pointer returned by **realloc** is not guaranteed to be the pointer passed through the *memblock* argument. **realloc** does not zero newly allocated memory in the case of buffer growth.

realloc sets **errno** to **ENOMEM** if the memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

realloc calls **malloc** in order to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **realloc** fails to allocate memory, **malloc** calls the new handler

routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1);
```

early in ones program, or link with NEWMODE.OBJ (see Link Options).

When the application is linked with a debug version of the C run-time libraries, **realloc** resolves to _realloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

realloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

ROUTINE	REQUIRED HEADER
realloc	<stdlib.h> and <malloc.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_realloc.c
// This program allocates a block of memory for
// buffer and then uses _msize to display the size of that
// block. Next, it uses realloc to expand the amount of
// memory used by buffer and then calls _msize again to
// display the new amount of memory allocated to buffer.

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main( void )
{
   long *buffer, *oldbuffer;
   size_t size;

   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );

   size = _msize( buffer );
   printf_s( "Size of block after malloc of 1000 longs: %u\n", size );

   // Reallocate and show new size:
   oldbuffer = buffer;     // save pointer in case realloc fails
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) ))
        ==  NULL )
   {
      free( oldbuffer );  // free original block
      exit( 1 );
   }
   size = _msize( buffer );
   printf_s( "Size of block after realloc of 1000 more longs: %u\n",
            size );

   free( buffer );
   exit( 0 );
}
```

```
Size of block after malloc of 1000 longs: 4000
Size of block after realloc of 1000 more longs: 8000
```

See also

Memory Allocation
calloc
free
malloc

_realloc_dbg

10/31/2018 • 2 minutes to read • Edit Online

Reallocates a specified block of memory in the heap by moving and/or resizing the block (debug version only).

Syntax

```
void *_realloc_dbg(
   void *userData,
   size_t newSize,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData
Pointer to the previously allocated memory block.

newSize
Requested size for the reallocated block (bytes).

blockType
Requested type for the reallocated block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to the name of the source file that requested the **realloc** operation or **NULL**.

linenumber
Line number in the source file where the **realloc** operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_realloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function either returns a pointer to the user portion of the reallocated memory block, calls the new handler function, or returns **NULL**. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the realloc function.

Remarks

_realloc_dbg is a debug version of the realloc function. When _DEBUG is not defined, each call to **_realloc_dbg** is reduced to a call to **realloc**. Both **realloc** and **_realloc_dbg** reallocate a memory block in the base heap, but **_realloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename*/*linenumber* information to determine the origin of allocation requests.

_realloc_dbg reallocates the specified memory block with slightly more space than the requested *newSize*. *newSize* might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug

header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. If the memory block is moved, the contents of the original block are overwritten.

_realloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails or if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the allocation block types and how they are used, see Types of blocks on the debug heap. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_realloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Example

See the example in the _msize_dbg topic.

See also

Debug Routines
_malloc_dbg

_recalloc

10/31/2018 • 2 minutes to read • Edit Online

A combination of **realloc** and **calloc**. Reallocates an array in memory and initializes its elements to 0.

Syntax

```
void *_recalloc(
   void *memblock
   size_t num,
   size_t size
);
```

Parameters

memblock
Pointer to previously allocated memory block.

number
Number of elements.

size
Length in bytes of each element.

Return Value

_recalloc returns a **void** pointer to the reallocated (and possibly moved) memory block.

If there is not enough available memory to expand the block to the given size, the original block is left unchanged, and **NULL** is returned.

If the requested size is zero, then the block pointed to by *memblock* is freed; the return value is **NULL**, and *memblock* is left pointing at a freed block.

The return value points to a storage space that is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **void**, use a type cast on the return value.

Remarks

The **_recalloc** function changes the size of an allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is **NULL**, **_recalloc** behaves the same way as calloc and allocates a new block of *number* * *size* bytes. Each element is initialized to 0. If *memblock* is not **NULL**, it should be a pointer returned by a previous call to **calloc**, malloc, or realloc.

Because the new block can be in a new memory location, the pointer returned by **_recalloc** is not guaranteed to be the pointer passed through the *memblock* argument.

_recalloc sets **errno** to **ENOMEM** if the memory allocation fails or if the amount of memory requested exceeds **_HEAP_MAXREQ**. For information on this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

recalloc calls **realloc** in order to use the C++ _set_new_mode function to set the new handler mode. The new handler mode indicates whether, on failure, **realloc** is to call the new handler routine as set by _set_new_handler.

By default, **realloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **_recalloc** fails to allocate memory, **realloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call

```
_set_new_mode(1);
```

early in the program, or link with NEWMODE.OBJ.

When the application is linked with a debug version of the C run-time libraries, **_recalloc** resolves to _recalloc_dbg. For more information about how the heap is managed during the debugging process, see The CRT Debug Heap.

_recalloc is marked `__declspec(noalias)` and `__declspec(restrict)`, meaning that the function is guaranteed not to modify global variables, and that the pointer returned is not aliased. For more information, see noalias and restrict.

Requirements

ROUTINE	REQUIRED HEADER
_recalloc	<stdlib.h> and <malloc.h>

For additional compatibility information, see Compatibility.

See also

Memory Allocation
_recalloc_dbg
_aligned_recalloc
_aligned_offset_recalloc
free
Link Options

_recalloc_dbg

10/31/2018 • 2 minutes to read • Edit Online

Reallocates an array and initializes its elements to 0 (debug version only).

Syntax

```
void *_recalloc_dbg(
   void *userData,
   size_t num,
   size_t size,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

userData
Pointer to the previously allocated memory block.

number
Requested number of memory blocks.

size
Requested size of each memory block (bytes).

blockType
Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

For information about the allocation block types and how they are used, see Types of blocks on the debug heap.

filename
Pointer to name of the source file that requested allocation operation or **NULL**.

linenumber
Line number in the source file where allocation operation was requested or **NULL**.

The *filename* and *linenumber* parameters are only available when **_recalloc_dbg** has been called explicitly or the _CRTDBG_MAP_ALLOC preprocessor constant has been defined.

Return Value

On successful completion, this function either returns a pointer to the user portion of the reallocated memory block, calls the new handler function, or returns **NULL**. For a complete description of the return behavior, see the following Remarks section. For more information about how the new handler function is used, see the _recalloc function.

Remarks

_recalloc_dbg is a debug version of the _recalloc function. When _DEBUG is not defined, each call to **_recalloc_dbg** is reduced to a call to **_recalloc**. Both **_recalloc** and **_recalloc_dbg** reallocate a memory block in the base heap, but **_recalloc_dbg** accommodates several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and

filename/*linenumber* information to determine the origin of allocation requests.

_recalloc_dbg reallocates the specified memory block with slightly more space than the requested size (*number* * *size*) which might be greater or less than the size of the originally allocated memory block. The additional space is used by the debug heap manager to link the debug memory blocks and to provide the application with debug header information and overwrite buffers. The reallocation might result in moving the original memory block to a different location in the heap, as well as changing the size of the memory block. The user portion of the block is filled with the value 0xCD and each of the overwrite buffers are filled with 0xFD.

_recalloc_dbg sets **errno** to **ENOMEM** if a memory allocation fails; **EINVAL** is returned if the amount of memory needed (including the overhead mentioned previously) exceeds **_HEAP_MAXREQ**. For information about this and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see CRT Debug Heap Details. For information about the differences between calling a standard heap function and its debug version in a debug build of an application, see Debug Versions of Heap Allocation Functions.

Requirements

ROUTINE	REQUIRED HEADER
_recalloc_dbg	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

See also

Debug Routines

remainder, remainderf, remainderl

10/31/2018 • 2 minutes to read • Edit Online

Computes the remainder of the quotient of two floating-point values, rounded to the nearest integral value.

Syntax

```
double remainder( double x, double y );
float remainderf( float x, float y );
long double remainderl( long double x, long double y );
```

```
float remainder( float x, float y ); /* C++ only */
long double remainder( long double x, long double y ); /* C++ only */
```

Parameters

x
The numerator.

y
The denominator.

Return Value

The floating-point remainder of *x* / *y*. If the value of *y* is 0.0, **remainder** returns a quiet NaN. For information about the representation of a quiet NaN by the **printf** family, see printf, _printf_l, wprintf, _wprintf_l.

Remarks

The **remainder** functions calculate the floating-point remainder *r* of *x* / *y* such that *x* = *n* * *y* + *r*, where *n* is the integer nearest in value to *x* / *y* and *n* is even whenever | *n* - *x* / *y* | = 1/2. When *r* = 0, *r* has the same sign as *x*.

Because C++ allows overloading, you can call overloads of **remainder** that take and return **float** or **long double** values. In a C program, **remainder** always takes two **double** arguments and returns a **double**.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
remainder, **remainderf**, **remainderl**	<math.h>	<cmath> or <math.h>

For compatibility information, see Compatibility.

Example

```
// crt_remainder.c
// This program displays a floating-point remainder.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double w = -10.0, x = 3.0, z;

   z = remainder(w, x);
   printf("The remainder of %.2f / %.2f is %f\n", w, x, z);
}
```

```
The remainder of -10.00 / 3.00 is -1.000000
```

See also

Floating-Point Support
ldiv, lldiv
imaxdiv
fmod, fmodf
remquo, remquof, remquol

remove, _wremove

1/21/2019 • 2 minutes to read • Edit Online

Delete a file.

Syntax

```
int remove(
   const char *path
);
int _wremove(
   const wchar_t *path
);
```

Parameters

path
Path of file to be removed.

Return Value

Each of these functions returns 0 if the file is successfully deleted. Otherwise, it returns -1 and sets **errno** either to **EACCES** to indicate that the path specifies a read-only file, specifies a directory, or the file is open, or to **ENOENT** to indicate that the filename or path was not found.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these and other return codes.

Remarks

The **remove** function deletes the file specified by *path*. **_wremove** is a wide-character version of **_remove**; the *path* argument to **_wremove** is a wide-character string. **_wremove** and **_remove** behave identically otherwise. All handles to a file must be closed before it can be deleted.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tremove	**remove**	**remove**	**_wremove**

Requirements

ROUTINE	REQUIRED HEADER
remove	<stdio.h> or <io.h>
_wremove	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_remove.c
/* This program uses remove to delete crt_remove.txt */

#include <stdio.h>

int main( void )
{
   if( remove( "crt_remove.txt" ) == -1 )
      perror( "Could not delete 'CRT_REMOVE.TXT'" );
   else
      printf( "Deleted 'CRT_REMOVE.TXT'\n" );
}
```

Input: crt_remove.txt

```
This file will be deleted.
```

Sample Output

```
Deleted 'CRT_REMOVE.TXT'
```

See also

File Handling
_unlink, _wunlink

remquo, remquof, remquol

10/31/2018 • 2 minutes to read • Edit Online

Computes the remainder of two integer values, and stores an integer value with the sign and approximate magnitude of the quotient in a location that's specified in a parameter.

Syntax

```
double remquo( double numer, double denom, int* quo );
float remquof( float numer, float denom, int* quo );
long double remquol( long double numer, long double denom, int* quo );
```

```
float remquo( float numer, float denom, int* quo ); /* C++ only */
long double remquo( long double numer, long double denom, int* quo ); /* C++ only */
```

Parameters

numer
The numerator.

denom
The denominator.

quo
A pointer to an integer to store a value that has the sign and approximate magnitude of the quotient.

Return Value

remquo returns the floating-point remainder of *x* / *y*. If the value of *y* is 0.0, **remquo** returns a quiet NaN. For information about the representation of a quiet NaN by the **printf** family, see printf, _printf_l, wprintf, _wprintf_l.

Remarks

The **remquo** function calculates the floating-point remainder *f* of *x* / *y* such that *x* = *i* * *y* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.

C++ allows overloading, so you can call overloads of **remquo** that take and return **float** or **long double** values. In a C program, **remquo** always takes two **double** arguments and returns a **double**.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
remquo, **remquof**, **remquol**	<math.h>	<cmath> or <math.h>

For compatibility information, see Compatibility.

Example

```
// crt_remquo.c
// This program displays a floating-point remainder.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double w = -10.0, x = 3.0, z;
   int quo = 0;

   z = remquo(w, x, &quo);
   printf("The remainder of %.2f / %.2f is %f\n", w, x, z);
   printf("Approximate signed quotient is %d\n", quo);
}
```

```
The remainder of -10.00 / 3.00 is -1.000000
Approximate signed quotient is -3
```

See also

Floating-Point Support
ldiv, lldiv
imaxdiv
fmod, fmodf
remainder, remainderf, remainderl

rename, _wrename

11/8/2018 • 2 minutes to read • Edit Online

Rename a file or directory.

Syntax

```
int rename(
   const char *oldname,
   const char *newname
);
int _wrename(
   const wchar_t *oldname,
   const wchar_t *newname
);
```

Parameters

oldname
Pointer to old name.

newname
Pointer to new name.

Return Value

Each of these functions returns 0 if it is successful. On an error, the function returns a nonzero value and sets **errno** to one of the following values:

ERRNO VALUE	CONDITION
EACCES	File or directory specified by *newname* already exists or could not be created (invalid path); or *oldname* is a directory and *newname* specifies a different path.
ENOENT	File or path specified by *oldname* not found.
EINVAL	Name contains invalid characters.

For other possible return values, see _doserrno, _errno, syserrlist, and _sys_nerr.

Remarks

The **rename** function renames the file or directory specified by *oldname* to the name given by *newname*. The old name must be the path of an existing file or directory. The new name must not be the name of an existing file or directory. You can use **rename** to move a file from one directory or device to another by giving a different path in the *newname* argument. However, you cannot use **rename** to move a directory. Directories can be renamed, but not moved.

_wrename is a wide-character version of **_rename**; the arguments to **_wrename** are wide-character strings. **_wrename** and **_rename** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_trename	**rename**	**rename**	**_wrename**

Requirements

ROUTINE	REQUIRED HEADER
rename	<io.h> or <stdio.h>
_wrename	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_renamer.c
/* This program attempts to rename a file named
* CRT_RENAMER.OBJ to CRT_RENAMER.JBO. For this operation
* to succeed, a file named CRT_RENAMER.OBJ must exist and
* a file named CRT_RENAMER.JBO must not exist.
*/

#include <stdio.h>

int main( void )
{
   int  result;
   char old[] = "CRT_RENAMER.OBJ", new[] = "CRT_RENAMER.JBO";

   /* Attempt to rename file: */
   result = rename( old, new );
   if( result != 0 )
      printf( "Could not rename '%s'\n", old );
   else
      printf( "File '%s' renamed to '%s'\n", old, new );
}
```

Output

```
File 'CRT_RENAMER.OBJ' renamed to 'CRT_RENAMER.JBO'
```

See also

File Handling

_resetstkoflw

10/31/2018 • 7 minutes to read • Edit Online

Recovers from stack overflow.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _resetstkoflw( void );
```

Return Value

Nonzero if the function succeeds, zero if it fails.

Remarks

The **_resetstkoflw** function recovers from a stack overflow condition, allowing a program to continue instead of failing with a fatal exception error. If the **_resetstkoflw** function is not called, there are no guard pages after the previous exception. The next time that there is a stack overflow, there are no exceptions at all and the process terminates without warning.

If a thread in an application causes an **EXCEPTION_STACK_OVERFLOW** exception, the thread has left its stack in a damaged state. This is in contrast to other exceptions such as **EXCEPTION_ACCESS_VIOLATION** or **EXCEPTION_INT_DIVIDE_BY_ZERO**, where the stack is not damaged. The stack is set to an arbitrarily small value when the program is first loaded. The stack then grows on demand to meet the needs of the thread. This is implemented by placing a page with PAGE_GUARD access at the end of the current stack. For more information, see Creating Guard Pages.

When the code causes the stack pointer to point to an address on this page, an exception occurs and the system does the following three things:

- Removes the PAGE_GUARD protection on the guard page so that the thread can read and write data to the memory.
- Allocates a new guard page that is located one page below the last one.
- Reruns the instruction that raised the exception.

In this way, the system can increase the size of the stack for the thread automatically. Each thread in a process has a maximum stack size. The stack size is set at compile time by the /STACK (Stack Allocations), or by the STACKSIZE statement in the .def file for the project.

When this maximum stack size is exceeded, the system does the following three things:

- Removes the PAGE_GUARD protection on the guard page, as previously described.
- Tries to allocate a new guard page below the last one. However, this fails because the maximum stack size

has been exceeded.

- Raises an exception so that the thread can handle it in the exception block.

Note that, at that point, the stack no longer has a guard page. The next time that the program grows the stack all the way to the end, where there should be a guard page, the program writes beyond the end of the stack and causes an access violation.

Call **_resetstkoflw** to restore the guard page whenever recovery is done after a stack overflow exception. This function can be called from inside the main body of an **__except** block or outside an **__except** block. However, there are some restrictions on when it should be used. **_resetstkoflw** should never be called from:

- A filter expression.
- A filter function.
- A function called from a filter function.
- A **catch** block.
- A **__finally** block.

At these points, the stack is not yet sufficiently unwound.

Stack overflow exceptions are generated as structured exceptions, not C++ exceptions, so **_resetstkoflw** is not useful in an ordinary **catch** block because it will not catch a stack overflow exception. However, if _set_se_translator is used to implement a structured exception translator that throws C++ exceptions (as in the second example), a stack overflow exception results in a C++ exception that can be handled by a C++ catch block.

It is not safe to call **_resetstkoflw** in a C++ catch block that is reached from an exception thrown by the structured exception translator function. In this case, the stack space is not freed and the stack pointer is not reset until outside the catch block, even though destructors have been called for any destructible objects before the catch block. This function should not be called until the stack space is freed and the stack pointer has been reset. Therefore, it should be called only after exiting the catch block. As little stack space as possible should be used in the catch block because a stack overflow that occurs in the catch block that is itself attempting to recover from a previous stack overflow is not recoverable and can cause the program to stop responding as the overflow in the catch block triggers an exception that itself is handled by the same catch block.

There are situations where **_resetstkoflw** can fail even if used in a correct location, such as within an **__except** block. If, even after unwinding the stack, there is still not enough stack space left to execute **_resetstkoflw** without writing into the last page of the stack, **_resetstkoflw** fails to reset the last page of the stack as the guard page and returns 0, indicating failure. Therefore, safe usage of this function should include checking the return value instead of assuming that the stack is safe to use.

Structured exception handling will not catch a **STATUS_STACK_OVERFLOW** exception when the application is compiled with **/clr** (see /clr (Common Language Runtime Compilation)).

Requirements

ROUTINE	REQUIRED HEADER
_resetstkoflw	<malloc.h>

For more compatibility information, see Compatibility.

Libraries: All versions of the CRT Library Features.

Example

The following example shows the recommended usage of the **_resetstkoflw** function.

```
// crt_resetstkoflw.c
// Launch program with and without arguments to observe
// the difference made by calling _resetstkoflw.

#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void recursive(int recurse)
{
   _alloca(2000);
   if (recurse)
      recursive(recurse);
}

// Filter for the stack overflow exception.
// This function traps the stack overflow exception, but passes
// all other exceptions through.
int stack_overflow_exception_filter(int exception_code)
{
   if (exception_code == EXCEPTION_STACK_OVERFLOW)
   {
       // Do not call _resetstkoflw here, because
       // at this point, the stack is not yet unwound.
       // Instead, signal that the handler (the __except block)
       // is to be executed.
       return EXCEPTION_EXECUTE_HANDLER;
   }
   else
       return EXCEPTION_CONTINUE_SEARCH;
}

int main(int ac)
{
   int i = 0;
   int recurse = 1, result = 0;

   for (i = 0 ; i < 10 ; i++)
   {
      printf("loop #%d\n", i + 1);
      __try
      {
         recursive(recurse);

      }

      __except(stack_overflow_exception_filter(GetExceptionCode()))
      {
         // Here, it is safe to reset the stack.

         if (ac >= 2)
         {
            puts("resetting stack overflow");
            result = _resetstkoflw();
         }
      }

      // Terminate if _resetstkoflw failed (returned 0)
      if (!result)
         return 3;
   }

   return 0;
}
```

Sample output with no program arguments:

```
loop #1
```

The program stops responding without executing further iterations.

With program arguments:

```
loop #1
resetting stack overflow
loop #2
resetting stack overflow
loop #3
resetting stack overflow
loop #4
resetting stack overflow
loop #5
resetting stack overflow
loop #6
resetting stack overflow
loop #7
resetting stack overflow
loop #8
resetting stack overflow
loop #9
resetting stack overflow
loop #10
resetting stack overflow
```

Description

The following example shows the recommended use of **_resetstkoflw** in a program where structured exceptions are converted to C++ exceptions.

Code

```
// crt_resetstkoflw2.cpp
// compile with: /EHa
// _set_se_translator requires the use of /EHa
#include <malloc.h>
#include <stdio.h>
#include <windows.h>
#include <eh.h>

class Exception { };

class StackOverflowException : Exception { };

// Because the overflow is deliberate, disable the warning that
// this function will cause a stack overflow.
#pragma warning (disable: 4717)
void CauseStackOverflow (int i)
{
    // Overflow the stack by allocating a large stack-based array
    // in a recursive function.
    int a[10000];
    printf("%d ", i);
    CauseStackOverflow (i + 1);
}

void __cdecl SEHTranslator (unsigned int code, _EXCEPTION_POINTERS*)
{
    // For stack overflow exceptions, throw our own C++
    // exception object.
```

```
    // For all other exceptions, throw a generic exception object.
    // Use minimal stack space in this function.
    // Do not call _resetstkoflw in this function.

    if (code == EXCEPTION_STACK_OVERFLOW)
        throw StackOverflowException ( );
    else
        throw Exception( );
}

int main ( )
{
    bool stack_reset = false;
    bool result = false;

    // Set up a function to handle all structured exceptions,
    // including stack overflow exceptions.
    _set_se_translator (SEHTranslator);

    try
    {
        CauseStackOverflow (0);
    }
    catch (StackOverflowException except)
    {
        // Use minimal stack space here.
        // Do not call _resetstkoflw here.
        printf("\nStack overflow!\n");
        stack_reset = true;
    }
    catch (Exception except)
    {
        // Do not call _resetstkoflw here.
        printf("\nUnknown Exception!\n");
    }
    if (stack_reset)
    {
        result = _resetstkoflw();
        // If stack reset failed, terminate the application.
        if (result == 0)
            exit(1);
    }

    void* pv = _alloca(100000);
    printf("Recovered from stack overflow and allocated 100,000 bytes"
           " using _alloca.");

    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Stack overflow!
Recovered from stack overflow and allocated 100,000 bytes using _alloca.
```

See also

_alloca

rewind

10/31/2018 • 2 minutes to read • Edit Online

Repositions the file pointer to the beginning of a file.

Syntax

```
void rewind(
   FILE *stream
);
```

Parameters

stream
Pointer to **FILE** structure.

Remarks

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. A call to **rewind** is similar to

(void) fseek(*stream***, 0L, SEEK_SET);**

However, unlike fseek, **rewind** clears the error indicators for the stream as well as the end-of-file indicator. Also, unlike fseek, **rewind** does not return a value to indicate whether the pointer was successfully moved.

To clear the keyboard buffer, use **rewind** with the stream **stdin**, which is associated with the keyboard by default.

If stream is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Requirements

ROUTINE	REQUIRED HEADER
rewind	<stdio.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_rewind.c
/* This program first opens a file named
* crt_rewind.out for input and output and writes two
* integers to the file. Next, it uses rewind to
* reposition the file pointer to the beginning of
* the file and reads the data back in.
*/
#include <stdio.h>

int main( void )
{
   FILE *stream;
   int data1, data2;

   data1 = 1;
   data2 = -37;

   fopen_s( &stream, "crt_rewind.out", "w+" );
   if( stream != NULL )
   {
      fprintf( stream, "%d %d", data1, data2 );
      printf( "The values written are: %d and %d\n", data1, data2 );
      rewind( stream );
      fscanf_s( stream, "%d %d", &data1, &data2 );
      printf( "The values read are: %d and %d\n", data1, data2 );
      fclose( stream );
   }
}
```

Output

```
The values written are: 1 and -37
The values read are: 1 and -37
```

See also

Stream I/O

rint, rintf, rintl

10/31/2018 • 2 minutes to read • Edit Online

Rounds a floating-point value to the nearest integer in floating-point format.

Syntax

```
double rint( double x );
float rintf( float x );
long double rintl( long double x );
```

```
float rint( float x );  // C++ only
long double rint( long double x );  // C++ only
```

Parameters

x
The floating-point value to round.

Return Value

The **rint** functions return a floating-point value that represents the nearest integer to *x*. Halfway values are rounded according to the current setting of the floating-point rounding mode, the same as the **nearbyint** functions. Unlike the **nearbyint** functions, the **rint** functions may raise the **FE_INEXACT** floating-point exception if the result differs in value from the argument. There is no error return.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± ∞, QNAN, IND	none	none
Denormals	EXCEPTION_FLT_UNDERFLOW	none

Remarks

Because C++ allows overloading, you can call overloads of **rint** that take and return **float** and **long double** values. In a C program, **rint** always takes and returns a **double**.

Requirements

FUNCTION	C HEADER	C++ HEADER
rint, **rintf**, **rintl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

Example

```
// crt_rint.c
// Build with: cl /W3 /Tc crt_rint.c
// This example displays the rounded results of
// the floating-point values 2.499999, -2.499999,
// 2.8, -2.8, 2.5 and -2.5.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.499999;
   float y = 2.8f;
   long double z = 2.5;

   printf("rint(%f) is %.0f\n", x, rint (x));
   printf("rint(%f) is %.0f\n", -x, rint (-x));
   printf("rintf(%f) is %.0f\n", y, rintf(y));
   printf("rintf(%f) is %.0f\n", -y, rintf(-y));
   printf("rintl(%Lf) is %.0Lf\n", z, rintl(z));
   printf("rintl(%Lf) is %.0Lf\n", -z, rintl(-z));
}
```

```
rint(2.499999) is 2
rint(-2.499999) is -2
rintf(2.800000) is 3
rintf(-2.800000) is -3
rintl(2.500000) is 3
rintl(-2.500000) is -3
```

See also

Floating-Point Support
ceil, ceilf, ceill
floor, floorf, floorl
fmod, fmodf
lrint, lrintf, lrintl, llrint, llrintf, llrintl
lround, lroundf, lroundl, llround, llroundf, llroundl
nearbyint, nearbyintf, nearbyintl
rint

rmdir

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _rmdir instead.

_rmdir, _wrmdir

11/8/2018 • 2 minutes to read • Edit Online

Deletes a directory.

Syntax

```
int _rmdir(
   const char *dirname
);
int _wrmdir(
   const wchar_t *dirname
);
```

Parameters

dirname
Path of the directory to be removed.

Return Value

Each of these functions returns 0 if the directory is successfully deleted. A return value of -1 indicates an error and **errno** is set to one of the following values:

ERRNO VALUE	CONDITION
ENOTEMPTY	Given path is not a directory, the directory is not empty, or the directory is either the current working directory or the root directory.
ENOENT	Path is invalid.
EACCES	A program has an open handle to the directory.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_rmdir** function deletes the directory specified by *dirname*. The directory must be empty, and it must not be the current working directory or the root directory.

_wrmdir is a wide-character version of **_rmdir**; the *dirname* argument to **_wrmdir** is a wide-character string. **_wrmdir** and **_rmdir** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_trmdir	**_rmdir**	**_rmdir**	**_wrmdir**

Requirements

ROUTINE	REQUIRED HEADER
_rmdir	<direct.h>
_wrmdir	<direct.h> or <wchar.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

See the example for _mkdir.

See also

Directory Control
_chdir, _wchdir
_mkdir, _wmkdir

rmtmp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _rmtmp instead.

_rmtmp

10/31/2018 • 2 minutes to read • Edit Online

Removes temporary files.

Syntax

```
int _rmtmp( void );
```

Return Value

_rmtmp returns the number of temporary files closed and deleted.

Remarks

The **_rmtmp** function cleans up all temporary files in the current directory. The function removes only those files created by **tmpfile**; use it only in the same directory in which the temporary files were created.

Requirements

ROUTINE	REQUIRED HEADER
_rmtmp	<stdio.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

See the example for tmpfile.

See also

Stream I/O
_flushall
tmpfile
_tempnam, _wtempnam, tmpnam, _wtmpnam

_rotl, _rotl64, _rotr, _rotr64

10/31/2018 • 2 minutes to read • Edit Online

Rotates bits to the left (**_rotl**) or right (**_rotr**).

Syntax

```
unsigned int _rotl(
   unsigned int value,
   int shift
);
unsigned __int64 _rotl64(
   unsigned __int64 value,
   int shift
);
unsigned int _rotr(
   unsigned int value,
   int shift
);
unsigned __int64 _rotr64(
   unsigned __int64 value,
   int shift
);
```

Parameters

value
Value to be rotated.

shift
Number of bits to shift.

Return Value

The rotated value. There is no error return.

Remarks

The **_rotl** and **_rotr** functions rotate the unsigned *value* by *shift* bits. **_rotl** rotates the value left. **_rotr** rotates the value right. Both functions wrap bits rotated off one end of *value* to the other end.

Requirements

ROUTINE	REQUIRED HEADER
_rotl, **_rotl64**	<stdlib.h>
_rotr, **_rotr64**	<stdlib.h>

For more compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_rot.c
/* This program shifts values to rotate an integer.
*/

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   unsigned val = 0x0fd93;
   __int64 val2 = 0x0101010101010101;

   printf( "0x%4.4x rotated left three times is 0x%4.4x\n",
           val, _rotl( val, 3 ) );
   printf( "0x%4.4x rotated right four times is 0x%4.4x\n",
           val, _rotr( val, 4 ) );

   printf( "%I64x rotated left three times is %I64x\n",
           val2, _rotl64( val2, 3 ) );
   printf( "%I64x rotated right four times is %I64x\n",
           val2, _rotr64( val2, 4 ) );
}
```

Output

```
0xfd93 rotated left three times is 0x7ec98
0xfd93 rotated right four times is 0x30000fd9
101010101010101 rotated left three times is 808080808080808
101010101010101 rotated right four times is 1010101010101010
```

See also

Floating-Point Support
_lrotl, _lrotr

round, roundf, roundl

10/31/2018 • 2 minutes to read • Edit Online

Rounds a floating-point value to the nearest integer.

Syntax

```
double round(
   double x
);
float round(
   float x
);  // C++ only
long double round(
   long double x
);  // C++ only
float roundf(
   float x
);
long double roundl(
   long double x
);
```

Parameters

x
The floating-point value to round.

Return Value

The **round** functions return a floating-point value that represents the nearest integer to *x*. Halfway values are rounded away from zero, regardless of the setting of the floating-point rounding mode. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± **QNAN**, **IND**	none	**_DOMAIN**

Remarks

Because C++ allows overloading, you can call overloads of **round** that take and return **float** and **long double** values. In a C program, **round** always takes and returns a **double**.

Requirements

ROUTINE	REQUIRED HEADER
round, **roundf**, **roundl**	<math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_round.c
// Build with: cl /W3 /Tc crt_round.c
// This example displays the rounded results of
// the floating-point values 2.499999, -2.499999,
// 2.8, -2.8, 2.5 and -2.5.

#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 2.499999;
   float y = 2.8f;
   long double z = 2.5;

   printf("round(%f) is %.0f\n", x, round(x));
   printf("round(%f) is %.0f\n", -x, round(-x));
   printf("roundf(%f) is %.0f\n", y, roundf(y));
   printf("roundf(%f) is %.0f\n", -y, roundf(-y));
   printf("roundl(%Lf) is %.0Lf\n", z, roundl(z));
   printf("roundl(%Lf) is %.0Lf\n", -z, roundl(-z));
}
```

```
round(2.499999) is 2
round(-2.499999) is -2
roundf(2.800000) is 3
roundf(-2.800000) is -3
roundl(2.500000) is 3
roundl(-2.500000) is -3
```

See also

Floating-Point Support
ceil, ceilf, ceill
floor, floorf, floorl
fmod, fmodf
lrint, lrintf, lrintl, llrint, llrintf, llrintl
lround, lroundf, lroundl, llround, llroundf, llroundl
nearbyint, nearbyintf, nearbyintl
rint, rintf, rintl

_RPT, _RPTF, _RPTW, _RPTFW Macros

10/31/2018 • 2 minutes to read • Edit Online

Tracks an application's progress by generating a debug report (debug version only). Note that *n* specifies the number of arguments in *args* and can be 0, 1, 2, 3, 4, or 5.

Syntax

```
_RPT
      n
      (
   reportType,
   format,
...[args]
);
_RPTFn(
   reportType,
   format,
   [args]
);
_RPTWn(
   reportType,
   format
   [args]
);
_RPTFWn(
   reportType,
   format
   [args]
);
```

Parameters

reportType
Report type: **_CRT_WARN**, **_CRT_ERROR**, or **_CRT_ASSERT**.

format
Format-control string used to create the user message.

args
Substitution arguments used by *format*.

Remarks

All these macros take the *reportType* and *format* parameters. In addition, they might also take up to four additional arguments, signified by the number appended to the macro name. For example, **_RPT0** and **_RPTF0** take no additional arguments, **_RPT1** and **_RPTF1** take *arg1*, **_RPT2** and **_RPTF2** take *arg1* and **arg2**, and so on.

The **_RPT** and **_RPTF** macros are similar to the printf function, because they can be used to track an application's progress during the debugging process. However, these macros are more flexible than **printf** because they do not need to be enclosed in **#ifdef** statements to prevent them from being called in a retail build of an application. This flexibility is achieved by using the _DEBUG macro; the **_RPT** and **_RPTF** macros are only available when the **_DEBUG** flag is defined. When **_DEBUG** is not defined, calls to these macros are removed during preprocessing.

The **_RPTW** and **_RPTFW** macros are wide-character versions of these macros. They are like **wprintf** and take wide-character strings as arguments.

The **_RPT** macros call the _CrtDbgReport function to generate a debug report with a user message. The **_RPTW** macros call the **_CrtDbgReportW** function to generate the same report with wide characters. The **_RPTF** and **_RPTFW** macros create a debug report with the source file and line number where the report macro was called, in addition to the user message. The user message is created by substituting the **arg**[*n*] arguments into the *format* string, using the same rules defined by the printf function.

_CrtDbgReport or **_CrtDbgReportW** generates the debug report and determines its destinations based on the current report modes and file defined for *reportType*. The _CrtSetReportMode and _CrtSetReportFile functions are used to define the destinations for each report type.

If an **_RPT** macro is called and neither **_CrtSetReportMode** nor **_CrtSetReportFile** has been called, messages are displayed as follows.

REPORT TYPE	OUTPUT DESTINATION
_CRT_WARN	Warning text is not displayed.
_CRT_ERROR	A pop-up window. Same as if `_CrtSetReportMode(_CRT_ERROR, _CRTDBG_MODE_WNDW);` had been specified.
_CRT_ASSERT	Same as **_CRT_ERROR**.

When the destination is a debug message window and the user chooses the **Retry** button, **_CrtDbgReport** or **_CrtDbgReportW** returns 1, causing these macros to start the debugger, provided that just-in-time (JIT) debugging is enabled. For more information about using these macros as a debugging error handling mechanism, see Using Macros for Verification and Reporting.

Two other macros exist that generate a debug report. The _ASSERT macro generates a report, but only when its expression argument evaluates to FALSE. _ASSERTE is exactly like **_ASSERT**, but includes the failed expression in the generated report.

Requirements

MACRO	REQUIRED HEADER
_RPT macros	<crtdbg.h>
_RPTF macros	<crtdbg.h>
_RPTW macros	<crtdbg.h>
_RPTFW macros	<crtdbg.h>

For more compatibility information, see Compatibility.

Libraries

Debug versions of C run-time libraries only.

Although these are macros and are obtained by including Crtdbg.h, the application must link with one of the debug libraries because these macros call other run-time functions.

Example

See the example in the _ASSERT topic.

See also

Debug Routines

_RTC_GetErrDesc

10/31/2018 • 2 minutes to read • Edit Online

Returns a brief description of a run-time error check (RTC) type.

Syntax

```
const char * _RTC_GetErrDesc(
   _RTC_ErrorNumber errnum
);
```

Parameters

errnum
A number between zero and one less than the value returned by **_RTC_NumErrors**.

Return Value

A character string that contains a short description of one of the error types detected by the run-time error check system. If error is less than zero or greater than or equal to the value returned by _RTC_NumErrors, **_RTC_GetErrDesc** returns **NULL**.

Requirements

ROUTINE	REQUIRED HEADER
_RTC_GetErrDesc	<rtcapi.h>

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

_RTC_NumErrors
Run-Time Error Checking

_RTC_NumErrors

11/9/2018 • 2 minutes to read • Edit Online

Returns the total number of errors that can be detected by run-time error checks (RTC). You can use this number as the control in a **for** loop, where each value in the loop is passed to _RTC_GetErrDesc.

Syntax

```
int _RTC_NumErrors( void );
```

Return Value

An integer whose value represents the total number of errors that can be detected by the Visual C++ run-time error checks.

Requirements

ROUTINE	REQUIRED HEADER
_RTC_NumErrors	<rtcapi.h>

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

_RTC_GetErrDesc
Run-Time Error Checking

_RTC_SetErrorFunc

10/31/2018 • 2 minutes to read • Edit Online

Designates a function as the handler for reporting run-time error checks (RTCs). This function is deprecated; use **_RTC_SetErrorFuncW** instead.

Syntax

```
_RTC_error_fn _RTC_SetErrorFunc(
   _RTC_error_fn function
);
```

Parameters

function
The address of the function that will handle run-time error checks.

Return Value

The previously defined error function. If there is no previously defined function, returns **NULL**.

Remarks

Do not use this function; instead, use **_RTC_SetErrorFuncW**. It is retained only for backward compatibility.

Requirements

ROUTINE	REQUIRED HEADER
_RTC_SetErrorFunc	<rtcapi.h>

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

_CrtDbgReport, _CrtDbgReportW
Run-Time Error Checking

_RTC_SetErrorFuncW

10/31/2018 • 2 minutes to read • Edit Online

Designates a function as the handler for the reporting of run-time error checks (RTCs).

Syntax

```
_RTC_error_fnW _RTC_SetErrorFuncW(
   _RTC_error_fnW function
);
```

Parameters

function
The address of the function that will handle run-time error checks.

Return Value

The previously defined error function; or **NULL** if there is no previously defined function.

Remarks

In new code, use only **_RTC_SetErrorFuncW**. **_RTC_SetErrorFunc** is only included in the library for backward compatibility.

The **_RTC_SetErrorFuncW** callback applies only to the component that it was linked in, but not globally.

Make sure that the address that you pass to **_RTC_SetErrorFuncW** is that of a valid error handling function.

If an error has been assigned a type of -1 by using _RTC_SetErrorType, the error handling function is not called.

Before you can call this function, you must first call one of the run-time error-check initialization functions. For more information, see Using Run-Time Checks Without the C Run-Time Library.

_RTC_error_fnW is defined as follows:

```
typedef int (__cdecl * _RTC_error_fnW)(
    int errorType,
    const wchar_t * filename,
    int linenumber,
    const wchar_t * moduleName,
    const wchar_t * format,
    ... );
```

where:

errorType
The type of error that's specified by _RTC_SetErrorType.

filename
The source file where the failure occurred, or null if no debug information is available.

linenumber
The line in *filename* where the failure occurred, or 0 if no debug information is available.

moduleName
The DLL or executable name where the failure occurred.

format
printf style string to display an error message, using the remaining parameters. The first argument of the VA_ARGLIST is the RTC Error number that occurred.

For an example that shows how to use **_RTC_error_fnW**, see Native Run-Time Checks Customization.

Requirements

ROUTINE	REQUIRED HEADER
_RTC_SetErrorFuncW	<rtcapi.h>

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

_CrtDbgReport, _CrtDbgReportW
Run-Time Error Checking

_RTC_SetErrorType

10/31/2018 • 2 minutes to read • Edit Online

Associates an error that is detected by run-time error checks (RTCs) with a type. Your error handler processes how to output errors of the specified type.

Syntax

```
int _RTC_SetErrorType(
   _RTC_ErrorNumber errnum,
   int ErrType
);
```

Parameters

errnum
A number between zero and one less than the value returned by _RTC_NumErrors.

ErrType
A value to assign to this *errnum*. For example, you might use **_CRT_ERROR**. If you are using **_CrtDbgReport** as your error handler, *ErrType* can only be one of the symbols defined in _CrtSetReportMode. If you have your own error handler (_RTC_SetErrorFunc), you can have as many *ErrType*s as there are *errnum*s.

An *ErrType* of _RTC_ERRTYPE_IGNORE has special meaning to **_CrtSetReportMode**; the error is ignored.

Return Value

The previous value for the error type *type*.

Remarks

By default, all errors are set to *ErrType* = 1, which corresponds to **_CRT_ERROR**. For more information about the default error types such as **_CRT_ERROR**, see _CrtDbgReport.

Before you can call this function, you must first call one of the run-time error check initialization functions; see Using Run-Time Checks without the C Run-Time Library

Requirements

ROUTINE	REQUIRED HEADER
_RTC_SetErrorType	<rtcapi.h>

For more information, see Compatibility.

Libraries

All versions of the C run-time libraries.

See also

_RTC_GetErrDesc
Run-Time Error Checking

_scalb, _scalbf

10/31/2018 • 2 minutes to read • Edit Online

Scales argument by a power of 2.

Syntax

```
double _scalb(
   double x,
   long exp
);
float _scalbf(
   float x,
   long exp
); /* x64 only */
```

Parameters

x
Double-precision, floating-point value.

exp
Long integer exponent.

Return Value

Returns an exponential value if successful. On overflow (depending on the sign of *x*), **_scalb** returns +/- **HUGE_VAL**; the **errno** variable is set to **ERANGE**.

For more information about this and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_scalb** function calculates the value of $x * 2^{exp}$.

Requirements

ROUTINE	REQUIRED HEADER
_scalb, **_scalbf**	<float.h>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
ldexp

scalbn, scalbnf, scalbnl, scalbln, scalblnf, scalblnl

10/31/2018 • 2 minutes to read • Edit Online

Multiplies a floating-point number by an integral power of FLT_RADIX.

Syntax

```
double scalbn(
   double x,
   int exp
);
float scalbn(
   float x,
   int exp
);  // C++ only
long double scalbn(
   long double x,
   int exp
);  // C++ only
float scalbnf(
   float x,
   int exp
);
long double scalbnl(
   long double x,
   int exp
);
double scalbln(
   double x,
   long exp
);
float scalbln(
   float x,
   long exp
);  // C++ only
long double scalbln(
   long double x,
   long exp
);  // C++ only
float scalblnf(
   float x,
   long exp
);
long double scalblnl(
   long double x,
   long exp
);
```

Parameters

x

Floating-point value.

exp

Integer exponent.

Return Value

The **scalbn** functions return the value of *x* * **FLT_RADIX**^{exp} when successful. On overflow (depending on the sign of *x*), **scalbn** returns +/- **HUGE_VAL**; the **errno** value is set to **ERANGE**.

For more information about **errno** and possible error return values, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

FLT_RADIX is defined in <float.h> as the native floating-point radix; on binary systems, it has a value of 2, and **scalbn** is equivalent to ldexp.

Because C++ allows overloading, you can call overloads of **scalbn** and **scalbln** that take and return **float** or **long double** types. In a C program, **scalbn** always takes a **double** and an **int** and returns a **double**, and **scalbln** always takes a **double** and a **long** and returns a **double**.

Requirements

FUNCTION	C HEADER	C++ HEADER
scalbn, **scalbnf**, **scalbnl**, **scalbln**, **scalblnf**, **scalblnl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

Example

```
// crt_scalbn.c
// Compile using: cl /W4 crt_scalbn.c
#include <math.h>
#include <stdio.h>

int main( void )
{
   double x = 6.4, y;
   int p = 3;

   y = scalbn( x, p );
   printf( "%2.1f times FLT_RADIX to the power of %d is %2.1f\n", x, p, y );
}
```

Output

```
6.4 times FLT_RADIX to the power of 3 is 51.2
```

See also

Floating-Point Support
frexp
ldexp
modf, modff, modfl

scanf, _scanf_l, wscanf, _wscanf_l

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data from the standard input stream. More secure versions of these function are available; see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l.

Syntax

```
int scanf(
   const char *format [,
   argument]...
);
int _scanf_l(
   const char *format,
   locale_t locale [,
   argument]...
);
int wscanf(
   const wchar_t *format [,
   argument]...
);
int _wscanf_l(
   const wchar_t *format,
   locale_t locale [,
   argument]...
);
```

Parameters

format
Format control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **scanf** function reads data from the standard input stream **stdin** and writes the data into the location given by *argument*. Each *argument* must be a pointer to a variable of a type that corresponds to a type specifier in *format*. If copying takes place between strings that overlap, the behavior is undefined.

> **IMPORTANT**
>
> When reading a string with **scanf**, always specify a width for the **%s** format (for example, **"%32s"** instead of **"%s"**); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, consider using scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l or fgets.

wscanf is a wide-character version of **scanf**; the *format* argument to **wscanf** is a wide-character string. **wscanf** and **scanf** behave identically if the stream is opened in ANSI mode. **scanf** doesn't currently support input from a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tscanf	**scanf**	**scanf**	**wscanf**
_tscanf_l	**_scanf_l**	**_scanf_l**	**_wscanf_l**

For more information, see Format Specification Fields — scanf functions and wscanf Functions.

Requirements

ROUTINE	REQUIRED HEADER
scanf, **_scanf_l**	<stdio.h>
wscanf, **_wscanf_l**	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_scanf.c
// compile with: /W3
// This program uses the scanf and wscanf functions
// to read formatted input.

#include <stdio.h>

int main( void )
{
   int   i, result;
   float fp;
   char  c, s[81];
   wchar_t wc, ws[81];
   result = scanf( "%d %f %c %C %80s %80S", &i, &fp, &c, &wc, s, ws ); // C4996
   // Note: scanf and wscanf are deprecated; consider using scanf_s and wscanf_s
   printf( "The number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c, wc, s, ws);
   result = wscanf( L"%d %f %hc %lc %80S %80ls", &i, &fp, &c, &wc, s, ws ); // C4996
   wprintf( L"The number of fields input is %d\n", result );
   wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp, c, wc, s, ws);
}
```

```
71 98.6 h z Byte characters
36 92.3 y n Wide characters
```

```
The number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

See also

Floating-Point Support
Stream I/O
Locale
fscanf, _fscanf_l, fwscanf, _fwscanf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sscanf, _sscanf_l, swscanf, _swscanf_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

3/27/2019 • 4 minutes to read • Edit Online

Reads formatted data from the standard input stream. These versions of scanf, _scanf_l, wscanf, _wscanf_l have security enhancements, as described in Security Features in the CRT.

Syntax

```
int scanf_s(
   const char *format [,
   argument]...
);
int _scanf_s_l(
   const char *format,
   locale_t locale [,
   argument]...
);
int wscanf_s(
   const wchar_t *format [,
   argument]...
);
int _wscanf_s_l(
   const wchar_t *format,
   locale_t locale [,
   argument]...
);
```

Parameters

format
Format control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of fields successfully converted and assigned. The return value doesn't include fields that were read but not assigned. A return value of 0 indicates no fields were assigned. The return value is **EOF** for an error, or if the end-of-file character or the end-of-string character is found in the first attempt to read a character. If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **scanf_s** and **wscanf_s** return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **scanf_s** function reads data from the standard input stream, **stdin**, and writes it into *argument*. Each *argument* must be a pointer to a variable type that corresponds to the type specifier in *format*. If copying occurs between strings that overlap, the behavior is undefined.

wscanf_s is a wide-character version of **scanf_s**; the *format* argument to **wscanf_s** is a wide-character string.

wscanf_s and **scanf_s** behave identically if the stream is opened in ANSI mode. **scanf_s** doesn't currently support input from a UNICODE stream.

The versions of these functions that have the **_l** suffix are identical, except they use the *locale* parameter instead of the current thread locale.

Unlike **scanf** and **wscanf**, **scanf_s** and **wscanf_s** require you to specify buffer sizes for some parameters. Specify the sizes for all **c**, **C**, **s**, **S**, or string control set **[]** parameters. The buffer size in characters is passed as an additional parameter. It immediately follows the pointer to the buffer or variable. For example, if you're reading a string, the buffer size for that string is passed as follows:

```
char s[10];
scanf_s("%9s", s, (unsigned)_countof(s)); // buffer size is 10, width specification is 9
```

The buffer size includes the terminal null. You can use a width specification field to ensure the token that's read in fits into the buffer. When a token is too large to fit, nothing is written to the buffer unless there's a width specification.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**. Use a static cast to convert a **size_t** value to **unsigned** for 64-bit build configurations.

The buffer size parameter describes the maximum number of characters, not bytes. In this example, the width of the buffer type doesn't match the width of the format specifier.

```
wchar_t ws[10];
wscanf_s(L"%9S", ws, (unsigned)_countof(ws));
```

The **S** format specifier means use the character width that's "opposite" the default width supported by the function. The character width is single byte, but the function supports double-byte characters. This example reads in a string of up to nine single-byte-wide characters and puts them in a double-byte-wide character buffer. The characters are treated as single-byte values; the first two characters are stored in `ws[0]`, the second two are stored in `ws[1]`, and so on.

This example reads a single character:

```
char c;
scanf_s("%c", &c, 1);
```

When multiple characters for non-null-terminated strings are read, integers are used for both the width specification and the buffer size.

```
char c[4];
scanf_s("%4c", c, (unsigned)_countof(c)); // not null terminated
```

For more information, see scanf Width Specification.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tscanf_s	**scanf_s**	**scanf_s**	**wscanf_s**
_tscanf_s_l	**_scanf_s_l**	**_scanf_s_l**	**_wscanf_s_l**

For more information, see Format Specification Fields: scanf and wscanf Functions.

Requirements

ROUTINE	REQUIRED HEADER
scanf_s, **_scanf_s_l**	<stdio.h>
wscanf_s, **_wscanf_s_l**	<stdio.h> or <wchar.h>

The console isn't supported in Universal Windows Platform (UWP) apps. The standard stream handles **stdin**, **stdout**, and **stderr** must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_scanf_s.c
// This program uses the scanf_s and wscanf_s functions
// to read formatted input.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int      i,
            result;
   float    fp;
   char     c,
            s[80];
   wchar_t  wc,
            ws[80];

   result = scanf_s( "%d %f %c %C %s %S", &i, &fp, &c, 1,
                     &wc, 1, s, (unsigned)_countof(s), ws, (unsigned)_countof(ws) );
   printf( "The number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c,
           wc, s, ws);
   result = wscanf_s( L"%d %f %hc %lc %S %ls", &i, &fp, &c, 2,
                      &wc, 1, s, (unsigned)_countof(s), ws, (unsigned)_countof(ws) );
   wprintf( L"The number of fields input is %d\n", result );
   wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp,
            c, wc, s, ws);
}
```

This program produces the following output when given this input:

```
71 98.6 h z Byte characters
36 92.3 y n Wide characters
```

```
The number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

See also

Floating-Point Support
Stream I/O
Locale
fscanf, _fscanf_l, fwscanf, _fwscanf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sscanf, _sscanf_l, swscanf, _swscanf_l

_scprintf, _scprintf_l, _scwprintf, _scwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the number of characters in the formatted string.

Syntax

```
int _scprintf(
   const char *format [,
   argument] ...
);
int _scprintf_l(
   const char *format,
   locale_t locale [,
   argument] ...
);
int _scwprintf(
   const wchar_t *format [,
   argument] ...
);
int _scwprintf_l(
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

Returns the number of characters that would be generated if the string were to be printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_scwprintf** performs the same function for wide characters.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in

instead of the current thread locale.

> **IMPORTANT**
>
> Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sctprintf	**_scprintf**	**_scprintf**	**_scwprintf**
_sctprintf_l	**_scprintf_l**	**_scprintf_l**	**_scwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
_scprintf, **_scprintf_l**	<stdio.h>
_scwprintf, **_scwprintf_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt__scprintf.c

#define _USE_MATH_DEFINES

#include <stdio.h>
#include <math.h>
#include <malloc.h>

int main( void )
{
   int count;
   int size;
   char *s = NULL;

   count = _scprintf( "The value of Pi is calculated to be %f.\n",
                      M_PI);

   size = count + 1; // the string will need one more char for the null terminator
   s = malloc(sizeof(char) * size);
   sprintf_s(s, size, "The value of Pi is calculated to be %f.\n",
                      M_PI);
   printf("The length of the following string will be %i.\n", count);
   printf("%s", s);
   free( s );
}
```

```
The length of the following string will be 46.
The value of Pi is calculated to be 3.141593.
```

See also

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the number of characters in the formatted string, with the ability to specify the order in which parameters are used in the format string.

Syntax

```
int _scprintf_p(
   const char *format [,
   argument] ...
);
int _scprintf_p_l(
   const char *format,
   locale_t locale [,
   argument] ...
);
int _scwprintf_p (
   const wchar_t *format [,
   argument] ...
);
int _scwprintf_p _l(
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
```

Parameters

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

Returns the number of characters that would be generated if the string were to be printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_scwprintf_p** performs the same function for wide characters.

The difference between **_scprintf_p** and **_scprintf** is that **_scprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sctprintf_p	**_scprintf_p**	**_scprintf_p**	**_scwprintf_p**
_sctprintf_p_l	**_scprintf_p_l**	**_scprintf_p_l**	**_scwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_scprintf_p, **_scprintf_p_l**	<stdio.h>
_scwprintf_p, **_scwprintf_p_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

See also

Stream I/O
_scprintf, _scprintf_l, _scwprintf, _scwprintf_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l

_searchenv, _wsearchenv

10/31/2018 • 2 minutes to read • Edit Online

Uses environment paths to search for a file. More secure versions of these functions are available; see _searchenv_s, _wsearchenv_s.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
void _searchenv(
   const char *filename,
   const char *varname,
   char *pathname
);
void _wsearchenv(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t *pathname
);
template <size_t size>
void _searchenv(
   const char *filename,
   const char *varname,
   char (&pathname)[size]
); // C++ only
template <size_t size>
void _wsearchenv(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t (&pathname)[size]
); // C++ only
```

Parameters

filename
Name of the file to search for.

varname
Environment to search.

pathname
Buffer to store the complete path.

Remarks

The **_searchenv** routine searches for the target file in the specified domain. The *varname* variable can be any environment or user-defined variable—for example, **PATH**, **LIB**, or **INCLUDE**—that specifies a list of directory paths. Because **_searchenv** is case-sensitive, *varname* should match the case of the environment variable.

The routine first searches for the file in the current working directory. If it does not find the file, it looks through the directories that are specified by the environment variable. If the target file is in one of those directories, the

newly created path is copied into *pathname*. If the *filename* file is not found, *pathname* contains an empty null-terminated string.

The *pathname* buffer should be at least **_MAX_PATH** characters long to accommodate the full length of the constructed path name. Otherwise, **_searchenv** might overrun the *pathname* buffer and cause unexpected behavior.

_wsearchenv is a wide-character version of **_searchenv**, and the arguments to **_wsearchenv** are wide-character strings. **_wsearchenv** and **_searchenv** behave identically otherwise.

If *filename* is an empty string, these functions return **ENOENT**.

If *filename* or *pathname* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For more information about **errno** and error codes, see errno Constants.

In C++, these functions have template overloads that invoke the newer, more secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsearchenv	**_searchenv**	**_searchenv**	**_wsearchenv**

Requirements

ROUTINE	REQUIRED HEADER
_searchenv	<stdlib.h>
_wsearchenv	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_searchenv.c
// compile with: /W3
// This program searches for a file in
// a directory that's specified by an environment variable.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";

   // Search for file in PATH environment variable:
   _searchenv( searchfile, envvar, pathbuffer ); // C4996
   // Note: _searchenv is deprecated; consider using _searchenv_s
   if( *pathbuffer != '\0' )
      printf( "Path for %s:\n%s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

```
Path for CL.EXE:
C:\Program Files\Microsoft Visual Studio 8\VC\BIN\CL.EXE
```

See also

Directory Control
getenv, _wgetenv
_putenv, _wputenv
_searchenv_s, _wsearchenv_s

_searchenv_s, _wsearchenv_s

10/31/2018 • 2 minutes to read • Edit Online

Searches for a file by using environment paths. These versions of _searchenv, _wsearchenv have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _searchenv_s(
   const char *filename,
   const char *varname,
   char *pathname,
   size_t numberOfElements
);
errno_t _wsearchenv_s(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t *pathname,
   size_t numberOfElements
);
template <size_t size>
errno_t _searchenv_s(
   const char *filename,
   const char *varname,
   char (&pathname)[size]
); // C++ only
template <size_t size>
errno_t _wsearchenv_s(
   const wchar_t *filename,
   const wchar_t *varname,
   wchar_t (&pathname)[size]
); // C++ only
```

Parameters

filename
Name of the file to search for.

varname
Environment to search.

pathname
Buffer to store the complete path.

numberOfElements
Size of the *pathname* buffer.

Return Value

Zero if successful; an error code on failure.

If *filename* is an empty string, the return value is **ENOENT**.

Error Conditions

FILENAME	*VARNAME*	*PATHNAME*	*NUMBEROFELEMENTS*	RETURN VALUE	CONTENTS OF *PATHNAME*
any	any	**NULL**	any	**EINVAL**	n/a
NULL	any	any	any	**EINVAL**	not changed
any	any	any	<= 0	**EINVAL**	not changed

If any of these error conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

The **_searchenv_s** routine searches for the target file in the specified domain. The *varname* variable can be any environment or user-defined variable that specifies a list of directory paths, such as **PATH**, **LIB**, and **INCLUDE**. Because **_searchenv_s** is case-sensitive, *varname* should match the case of the environment variable. If *varname* does not match the name of an environment variable defined in the process's environment, the function returns zero and the *pathname* variable is unchanged.

The routine searches first for the file in the current working directory. If it does not find the file, it looks next through the directories specified by the environment variable. If the target file is in one of those directories, the newly created path is copied into *pathname*. If the *filename* file is not found, *pathname* contains an empty null-terminated string.

The *pathname* buffer should be at least **_MAX_PATH** characters long to accommodate the full length of the constructed path name. Otherwise, **_searchenv_s** might overrun the *pathname* buffer resulting in unexpected behavior.

_wsearchenv_s is a wide-character version of **_searchenv_s**; the arguments to **_wsearchenv_s** are wide-character strings. **_wsearchenv_s** and **_searchenv_s** behave identically otherwise.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsearchenv_s	**_searchenv_s**	**_searchenv_s**	**_wsearchenv_s**

Requirements

ROUTINE	REQUIRED HEADER
_searchenv_s	<stdlib.h>
_wsearchenv_s	<stdlib.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_searchenv_s.c
/* This program searches for a file in
* a directory specified by an environment variable.
*/

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";
   errno_t err;

   /* Search for file in PATH environment variable: */
   err = _searchenv_s( searchfile, envvar, pathbuffer, _MAX_PATH );
   if (err != 0)
   {
      printf("Error searching the path. Error code: %d\n", err);
   }
   if( *pathbuffer != '\0' )
      printf( "Path for %s:\n%s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

```
Path for CL.EXE:
C:\Program Files\Microsoft Visual Studio 2010\VC\BIN\CL.EXE
```

See also

Directory Control
_searchenv, _wsearchenv
getenv, _wgetenv
_putenv, _wputenv

__security_init_cookie

10/31/2018 • 2 minutes to read • Edit Online

Initializes the global security cookie.

Syntax

```
void __security_init_cookie(void);
```

Remarks

The global security cookie is used for buffer overrun protection in code compiled with /GS (Buffer Security Check) and in code that uses exception handling. On entry to an overrun-protected function, the cookie is put on the stack, and on exit, the value on the stack is compared with the global cookie. Any difference between them indicates that a buffer overrun has occurred and causes immediate termination of the program.

Normally, **__security_init_cookie** is called by the CRT when it is initialized. If you bypass CRT initialization—for example, if you use /ENTRY to specify an entry-point—then you must call **__security_init_cookie** yourself. If **__security_init_cookie** is not called, the global security cookie is set to a default value and buffer overrun protection is compromised. Because an attacker can exploit this default cookie value to defeat the buffer overrun checks, we recommend that you always call **__security_init_cookie** when you define your own entry point.

The call to **__security_init_cookie** must be made before any overrun-protected function is entered; otherwise a spurious buffer overrun will be detected. For more information, see C Runtime Error R6035.

Example

See the examples in C Runtime Error R6035.

Requirements

ROUTINE	REQUIRED HEADER
__security_init_cookie	<process.h>

__security_init_cookie is a Microsoft extension to the standard C Runtime Library. For compatibility information, see Compatibility.

See also

Microsoft Security Response Center

_seh_filter_dll, _seh_filter_exe

10/31/2018 • 2 minutes to read • Edit Online

Identifies the exception and the related action to be taken.

Syntax

```
int __cdecl _seh_filter_dll(
   unsigned long _ExceptionNum,
   struct _EXCEPTION_POINTERS* _ExceptionPtr
);
int __cdecl _seh_filter_exe(
   unsigned long _ExceptionNum,
   struct _EXCEPTION_POINTERS* _ExceptionPtr
);
```

Parameters

_ExceptionNum
The identifier for the exception.

_ExceptionPtr
A pointer to the exception information.

Return Value

An integer that indicates the action to be taken, based on the result of exception processing.

Remarks

These methods are called by the exception-filter expression of the try-except Statement. The method consults a constant internal table to identify the exception and determine the appropriate action, as shown here. The exception numbers are defined in winnt.h and the signal numbers are defined in signal.h.

EXCEPTION NUMBER (UNSIGNED LONG)	SIGNAL NUMBER
STATUS_ACCESS_VIOLATION	SIGSEGV
STATUS_ILLEGAL_INSTRUCTION	SIGILL
STATUS_PRIVILEGED_INSTRUCTION	SIGILL
STATUS_FLOAT_DENORMAL_OPERAND	SIGFPE
STATUS_FLOAT_DIVIDE_BY_ZERO	SIGFPE
STATUS_FLOAT_INEXACT_RESULT	SIGFPE
STATUS_FLOAT_INVALID_OPERATION	SIGFPE
STATUS_FLOAT_OVERFLOW	SIGFPE

EXCEPTION NUMBER (UNSIGNED LONG)	SIGNAL NUMBER
STATUS_FLOAT_STACK_CHECK	SIGFPE
STATUS_FLOAT_UNDERFLOW	SIGFPE

Requirements

Header: corecrt_startup.h

See also

Alphabetical Function Reference

_set_abort_behavior

10/31/2018 • 2 minutes to read • Edit Online

Specifies the action to be taken when a program is abnormally terminated.

> **NOTE**
> Do not use the abort function to shut down a Microsoft Store app, except in testing or debugging scenarios. Programmatic or UI ways to close a Store app are not permitted according to the Microsoft Store policies. For more information, see UWP app lifecycle.

Syntax

```
unsigned int _set_abort_behavior(
   unsigned int flags,
   unsigned int mask
);
```

Parameters

flags
New value of the abort flags.

mask
Mask for the abort flags bits to set.

Return Value

The old value of the flags.

Remarks

There are two abort flags: **_WRITE_ABORT_MSG** and **_CALL_REPORTFAULT**. **_WRITE_ABORT_MSG** determines whether a helpful text message is printed when a program is abnormally terminated. The message states that the application has called the abort function. The default behavior is to print the message. **_CALL_REPORTFAULT**, if set, specifies that a Watson crash dump is generated and reported when abort is called. By default, crash dump reporting is enabled in non-DEBUG builds.

Requirements

ROUTINE	REQUIRED HEADER
_set_abort_behavior	<stdlib.h>

For more compatibility information, see Compatibility.

Example

```
// crt_set_abort_behavior.c
// compile with: /TC
#include <stdlib.h>

int main()
{
   printf("Suppressing the abort message. If successful, this message"
          " will be the only output.\n");
   // Suppress the abort message
   _set_abort_behavior( 0, _WRITE_ABORT_MSG);
   abort();
}
```

```
Suppressing the abort message. If successful, this message will be the only output.
```

See also

abort

setbuf

4/9/2019 • 2 minutes to read • Edit Online

Controls stream buffering. This function is deprecated; use setvbuf instead.

Syntax

```
void setbuf(
   FILE *stream,
   char *buffer
);
```

Parameters

stream
Pointer to **FILE** structure.

buffer
User-allocated buffer.

Remarks

The **setbuf** function controls buffering for *stream*. The *stream* argument must refer to an open file that hasn't been read or written. If the *buffer* argument is **NULL**, the stream is unbuffered. If not, the buffer must point to a character array of length **BUFSIZ**, where **BUFSIZ** is the buffer size as defined in STDIO.H. The user-specified buffer, instead of the default system-allocated buffer for the given stream, is used for I/O buffering. The **stderr** stream is unbuffered by default, but you can use **setbuf** to assign buffers to **stderr**.

setbuf has been replaced by setvbuf, which is the preferred routine for new code. Unlike **setvbuf**, **setbuf** has no way of reporting errors. **setvbuf** also lets you control both the buffering mode and the buffer size. **setbuf** exists for compatibility with existing code.

Requirements

ROUTINE	REQUIRED HEADER
setbuf	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_setbuf.c
// compile with: /W3
// This program first opens files named DATA1 and
// DATA2. Then it uses setbuf to give DATA1 a user-assigned
// buffer and to change DATA2 so that it has no buffer.

#include <stdio.h>

int main( void )
{
   char buf[BUFSIZ];
   FILE *stream1, *stream2;

   fopen_s( &stream1, "data1", "a" );
   fopen_s( &stream2, "data2", "w" );

   if( (stream1 != NULL) && (stream2 != NULL) )
   {
      // "stream1" uses user-assigned buffer:
      setbuf( stream1, buf ); // C4996
      // Note: setbuf is deprecated; consider using setvbuf instead
      printf( "stream1 set to user-defined buffer at: %Fp\n", buf );

      // "stream2" is unbuffered
      setbuf( stream2, NULL ); // C4996
      printf( "stream2 buffering disabled\n" );
      _fcloseall();
   }
}
```

```
stream1 set to user-defined buffer at: 0012FCDC
stream2 buffering disabled
```

See also

Stream I/O
fclose, _fcloseall
fflush
fopen, _wfopen
setvbuf

_set_controlfp

10/31/2018 • 2 minutes to read • Edit Online

Sets the floating-point control word.

Syntax

```
void __cdecl _set_controlfp(
    unsigned int newControl,
    unsigned int mask
);
```

Parameters

newControl
New control-word bit values.

mask
Mask for new control-word bits to set.

Return Value

None.

Remarks

The **_set_controlfp** function is similar to **_control87**, but it only sets the floating-point control word to *newControl*. The bits in the values indicate the floating-point control state. The floating-point control state allows the program to change the precision, rounding, and infinity modes in the floating-point math package. You can also mask or unmask floating-point exceptions using **_set_controlfp**. For more information, see _control87, _controlfp, __control87_2.

This function is deprecated when compiling with /clr (Common Language Runtime Compilation) because the common language runtime only supports the default floating-point precision.

Requirements

ROUTINE	REQUIRED HEADER	COMPATIBILITY
_set_controlfp	<float.h>	x86 processor only

For more compatibility information, see Compatibility.

See also

Floating-Point Support
_clear87, _clearfp
_status87, _statusfp, _statusfp2

_set_doserrno

11/8/2018 • 2 minutes to read • Edit Online

Sets the value of the _doserrno global variable.

Syntax

```
errno_t _set_doserrno( int error_value );
```

Parameters

error_value
The new value of **_doserrno**.

Return Value

Returns zero if successful.

Remarks

Possible values are defined in Errno.h.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_set_doserrno	<stdlib.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

_get_doserrno
errno, _doserrno, _sys_errlist, and _sys_nerr

_set_errno

11/8/2018 • 2 minutes to read • Edit Online

Set the value of the **errno** global variable.

Syntax

```
errno_t _set_errno( int error_value );
```

Parameters

error_value
The new value of **errno**.

Return Value

Returns zero if successful.

Remarks

Possible values are defined in Errno.h. Also, see errno Constants.

Example

```
// crt_set_errno.c
#include <stdio.h>
#include <errno.h>

int main()
{
   _set_errno( EILSEQ );
   perror( "Oops" );
}
```

```
Oops: Illegal byte sequence
```

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_set_errno	<stdlib.h>	<errno.h>

For more compatibility information, see Compatibility.

See also

_get_errno
errno, _doserrno, _sys_errlist, and _sys_nerr

_set_error_mode

10/31/2018 • 2 minutes to read • Edit Online

Modifies **__error_mode** to determine a non-default location where the C runtime writes an error message for an error that might end the program.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _set_error_mode(
   int mode_val
);
```

Parameters

mode_val
Destination of error messages.

Return Value

Returns the old setting or -1 if an error occurs.

Remarks

Controls the error output sink by setting the value of **__error_mode**. For example, you can direct output to a standard error or use the **MessageBox** API.

The *mode_val* parameter can be set to one of the following values.

PARAMETER	DESCRIPTION
_OUT_TO_DEFAULT	Error sink is determined by **__app_type**.
_OUT_TO_STDERR	Error sink is a standard error.
_OUT_TO_MSGBOX	Error sink is a message box.
_REPORT_ERRMODE	Report the current **__error_mode** value.

If a value other than those listed is passed in, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_set_error_mode** sets **errno** to **EINVAL** and returns -1.

When it's used with an assert, **_set_error_mode** displays the failed statement in the dialog box and gives you the option of choosing the **Ignore** button so that you can continue to run the program.

Requirements

ROUTINE	REQUIRED HEADER
_set_error_mode	<stdlib.h>

Example

```
// crt_set_error_mode.c
// compile with: /c
#include <stdlib.h>
#include <assert.h>

int main()
{
   _set_error_mode(_OUT_TO_STDERR);
   assert(2+2==5);
}
```

```
Assertion failed: 2+2==5, file crt_set_error_mode.c, line 8

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

See also

assert Macro, _assert, _wassert

_set_fmode

11/8/2018 • 2 minutes to read • Edit Online

Sets the default file translation mode for file I/O operations.

Syntax

```
errno_t _set_fmode(
   int mode
);
```

Parameters

mode
The file translation mode desired: **_O_TEXT** or **_O_BINARY**.

Return Value

Returns zero if successful, an error code on failure. If *mode* is not **_O_TEXT** or **_O_BINARY** or **_O_WTEXT**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **EINVAL**.

Remarks

The function sets the _fmode global variable. This variable specifies the default file translation mode for the file I/O operations **_open** and **_pipe**.

_O_TEXT and **_O_BINARY** are defined in Fcntl.h. **EINVAL** is defined in Errno.h.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_set_fmode	<stdlib.h>	<fcntl.h>, <errno.h>

For more compatibility information, see Compatibility.

Example

```
// crt_set_fmode.c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>     /* for _O_TEXT and _O_BINARY */
#include <errno.h>     /* for EINVAL */
#include <sys\stat.h>  /* for _S_IWRITE */
#include <share.h>     /* for _SH_DENYNO */

int main()
{
   int mode, fd, ret;
   errno_t err;
   int buf[12] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
                   75, 76 };
   char * filename = "fmode.out";

   err = _get_fmode(&mode);
   if (err == EINVAL)
   {
      printf( "Invalid parameter: mode\n");
      return 1;
   }
   else
      printf( "Default Mode is %s\n", mode == _O_TEXT ? "text" :
              "binary");

   err = _set_fmode(_O_BINARY);
   if (err == EINVAL)
   {
      printf( "Invalid mode.\n");
      return 1;
   }

   if ( _sopen_s(&fd, filename, _O_RDWR | _O_CREAT, _SH_DENYNO, _S_IWRITE | _S_IREAD) != 0 )
   {
      printf( "Error opening the file %s\n", filename);
      return 1;
   }

   if (ret = _write(fd, buf, 12*sizeof(int)) < 12*sizeof(int))
   {
      printf( "Problem writing to the file %s.\n", filename);
      printf( "Number of bytes written: %d\n", ret);
   }

   if (_close(fd) != 0)
   {
      printf("Error closing the file %s. Error code %d.\n",
             filename, errno);
   }

   system("type fmode.out");
}
```

```
Default Mode is binary
A   B   C   D   E   F   G   H   I   J   K   L
```

See also

_fmode
_get_fmode
_setmode
Text and Binary Mode File I/O

_set_invalid_parameter_handler, _set_thread_local_invalid_parameter_handler

10/31/2018 • 2 minutes to read • Edit Online

Sets a function to be called when the CRT detects an invalid argument.

Syntax

```
_invalid_parameter_handler _set_invalid_parameter_handler(
   _invalid_parameter_handler pNew
);
_invalid_parameter_handler _set_thread_local_invalid_parameter_handler(
   _invalid_parameter_handler pNew
);
```

Parameters

pNew
The function pointer to the new invalid parameter handler.

Return Value

A pointer to the invalid parameter handler before the call.

Remarks

Many C runtime functions check the validity of arguments passed to them. If an invalid argument is passed, the function can set the **errno** error number or return an error code. In such cases, the invalid parameter handler is also called. The C runtime supplies a default global invalid parameter handler that terminates the program and displays a runtime error message. You can use the **_set_invalid_parameter_handler** to set your own function as the global invalid parameter handler. The C runtime also supports a thread-local invalid parameter handler. If a thread-local parameter handler is set in a thread by using **_set_thread_local_invalid_parameter_handler**, the C runtime functions called from the thread use that handler instead of the global handler. Only one function can be specified as the global invalid argument handler at a time. Only one function can be specified as the thread-local invalid argument handler per thread, but different threads can have different thread-local handlers. This allows you to change the handler used in one part of your code without affecting the behavior of other threads.

When the runtime calls the invalid parameter function, it usually means that a nonrecoverable error occurred. The invalid parameter handler function you supply should save any data it can and then abort. It should not return control to the main function unless you are confident that the error is recoverable.

The invalid parameter handler function must have the following prototype:

```
void _invalid_parameter(
   const wchar_t * expression,
   const wchar_t * function,
   const wchar_t * file,
   unsigned int line,
   uintptr_t pReserved
);
```

The *expression* argument is a wide string representation of the argument expression that raised the error. The *function* argument is the name of the CRT function that received the invalid argument. The *file* argument is the name of the CRT source file that contains the function. The *line* argument is the line number in that file. The last argument is reserved. The parameters all have the value **NULL** unless a debug version of the CRT library is used.

Requirements

ROUTINE	REQUIRED HEADER
_set_invalid_parameter_handler, **_set_thread_local_invalid_parameter_handler**	C: <stdlib.h> C++: <cstdlib> or <stdlib.h>

The **_set_invalid_parameter_handler** and **_set_thread_local_invalid_parameter_handler** functions are Microsoft specific. For compatibility information, see Compatibility.

Example

In the following example, an invalid parameter error handler is used to print the function that received the invalid parameter and the file and line in CRT sources. When the debug CRT library is used, invalid parameter errors also raise an assertion, which is disabled in this example using _CrtSetReportMode.

```
// crt_set_invalid_parameter_handler.c
// compile with: /Zi /MTd
#include <stdio.h>
#include <stdlib.h>
#include <crtdbg.h>  // For _CrtSetReportMode

void myInvalidParameterHandler(const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf(L"Invalid parameter detected in function %s."
            L" File: %s Line: %d\n", function, file, line);
   wprintf(L"Expression: %s\n", expression);
   abort();
}

int main( )
{
   char* formatString;

   _invalid_parameter_handler oldHandler, newHandler;
   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);

   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   // Call printf_s with invalid parameters.

   formatString = NULL;
   printf(formatString);
}
```

```
Invalid parameter detected in function common_vfprintf. File:
minkernel\crts\ucrt\src\appcrt\stdio\output.cpp Line: 32
Expression: format != nullptr
```

See also

_get_invalid_parameter_handler, _get_thread_local_invalid_parameter_handler
Security-Enhanced Versions of CRT Functions
errno, _doserrno, _sys_errlist, and _sys_nerr

setjmp

3/1/2019 • 2 minutes to read • Edit Online

Saves the current state of the program.

Syntax

```
int setjmp(
   jmp_buf env
);
```

Parameters

env
Variable in which environment is stored.

Return Value

Returns 0 after saving the stack environment. If **setjmp** returns as a result of a `longjmp` call, it returns the *value* argument of `longjmp`, or if the *value* argument of `longjmp` is 0, **setjmp** returns 1. There is no error return.

Remarks

The **setjmp** function saves a stack environment, which you can subsequently restore, using `longjmp`. When used together, **setjmp** and `longjmp` provide a way to execute a non-local **goto**. They are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** saves the current stack environment in *env*. A subsequent call to `longjmp` restores the saved environment and returns control to the point just after the corresponding **setjmp** call. All variables (except register variables) accessible to the routine receiving control contain the values they had when `longjmp` was called.

It is not possible to use **setjmp** to jump from native to managed code.

Microsoft Specific

In Microsoft C++ code on Windows, **longjmp** uses the same stack-unwinding semantics as exception-handling code. It is safe to use in the same places that C++ exceptions can be raised. However, this usage is not portable, and comes with some important caveats. For details, see longjmp.

END Microsoft Specific

> **NOTE**
> In portable C++ code, you can't assume `setjmp` and `longjmp` support C++ object semantics. Specifically, a `setjmp` / `longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by **catch** and **throw** would invoke any non-trivial destructors for any automatic objects. In C++ programs, we recommend you use the C++ exception-handling mechanism.

For more information, see Using setjmp and longjmp.

Requirements

ROUTINE	REQUIRED HEADER
setjmp	<setjmp.h>

For additional compatibility information, see Compatibility.

Example

See the example for _fpreset.

See also

Process and Environment Control
longjmp

setlocale, _wsetlocale

3/22/2019 • 8 minutes to read • Edit Online

Sets or retrieves the run-time locale.

Syntax

```
char *setlocale(
   int category,
   const char *locale
);
wchar_t *_wsetlocale(
   int category,
   const wchar_t *locale
);
```

Parameters

category
Category affected by locale.

locale
Locale specifier.

Return Value

If a valid *locale* and *category* are given, returns a pointer to the string associated with the specified *locale* and *category*. If the *locale* or *category* is not valid, returns a null pointer and the current locale settings of the program are not changed.

For example, the call

```
setlocale( LC_ALL, "en-US" );
```

sets all categories, returning only the string

```
en-US
```

You can copy the string returned by **setlocale** to restore that part of the program's locale information. Global or thread local storage is used for the string returned by **setlocale**. Later calls to **setlocale** overwrite the string, which invalidates string pointers returned by earlier calls.

Remarks

Use the **setlocale** function to set, change, or query some or all of the current program locale information specified by *locale* and *category*. *locale* refers to the locality (country/region and language) for which you can customize certain aspects of your program. Some locale-dependent categories include the formatting of dates and the display format for monetary values. If you set *locale* to the default string for a language that has multiple forms supported on your computer, you should check the **setlocale**

return value to see which language is in effect. For example, if you set *locale* to "chinese" the return value could be either "chinese-simplified" or "chinese-traditional".

_wsetlocale is a wide-character version of **setlocale**; the *locale* argument and return value of **_wsetlocale** are wide-character strings. **_wsetlocale** and **setlocale** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsetlocale	**setlocale**	**setlocale**	**_wsetlocale**

The *category* argument specifies the parts of a program's locale information that are affected. The macros used for *category* and the parts of the program they affect are as follows:

CATEGORY FLAG	AFFECTS
LC_ALL	All categories, as listed below.
LC_COLLATE	The **strcoll**, **_stricoll**, **wcscoll**, **_wcsicoll**, **strxfrm**, **_strncoll**, **_strnicoll**, **_wcsncoll**, **_wcsnicoll**, and **wcsxfrm** functions.
LC_CTYPE	The character-handling functions (except **isdigit**, **isxdigit**, **mbstowcs**, and **mbtowc**, which are unaffected).
LC_MONETARY	Monetary-formatting information returned by the **localeconv** function.
LC_NUMERIC	Decimal-point character for the formatted output routines (such as **printf**), for the data-conversion routines, and for the non-monetary formatting information returned by **localeconv**. In addition to the decimal-point character, **LC_NUMERIC** sets the thousands separator and the grouping control string returned by localeconv.
LC_TIME	The **strftime** and **wcsftime** functions.

This function validates the category parameter. If the category parameter is not one of the values given in the previous table, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function sets **errno** to **EINVAL** and returns **NULL**.

The *locale* argument is a pointer to a string that specifies the locale. For information about the format of the *locale* argument, see Locale Names, Languages, and Country/Region Strings. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of **C** specifies the minimal ANSI conforming environment for C translation. The **C** locale assumes that all **char** data types are 1 byte and that their value is always less than 256.

At program startup, the equivalent of the following statement is executed:

`setlocale( LC_ALL, "C" );`

The *locale* argument can take a locale name, a language string, a language string and country/region code, a code page, or a language string, country/region code, and code page. The set of available locale names, languages, country/region codes, and code pages includes all those supported by the Windows NLS API except code pages that require more than two bytes per character, such as UTF-7 and UTF-8. If you provide a code page value of UTF-7 or UTF-8, **setlocale** will fail, returning **NULL**. The set of locale names supported by **setlocale** are described in Locale Names, Languages, and Country/Region Strings. The set of language and country/region strings supported by **setlocale** are listed in Language Strings and Country/Region Strings. We recommend the locale name form for performance and for maintainability of locale strings embedded in code or serialized to storage. The locale name strings are less likely to be changed by an operating system update than the language and country/region name form.

A null pointer that's passed as the *locale* argument tells **setlocale** to query instead of to set the international environment. If the *locale* argument is a null pointer, the program's current locale setting is not changed. Instead, **setlocale** returns a pointer to the string that's associated with the *category* of the thread's current locale. If the *category* argument is **LC_ALL**, the function returns a string that indicates the current setting of each category, separated by semicolons. For example, the sequence of calls

```
// Set all categories and return "en-US"
setlocale(LC_ALL, "en-US");
// Set only the LC_MONETARY category and return "fr-FR"
setlocale(LC_MONETARY, "fr-FR");
printf("%s\n", setlocale(LC_ALL, NULL));
```

returns

```
LC_COLLATE=en-US;LC_CTYPE=en-US;LC_MONETARY=fr-FR;LC_NUMERIC=en-US;LC_TIME=en-US
```

which is the string that's associated with the **LC_ALL** category.

The following examples pertain to the **LC_ALL** category. Either of the strings ".OCP" and ".ACP" can be used instead of a code page number to specify use of the user-default OEM code page and user-default ANSI code page, respectively.

- `setlocale( LC_ALL, "" );`

  Sets the locale to the default, which is the user-default ANSI code page obtained from the operating system.

- `setlocale( LC_ALL, ".OCP" );`

  Explicitly sets the locale to the current OEM code page obtained from the operating system.

- `setlocale( LC_ALL, ".ACP" );`

  Sets the locale to the ANSI code page obtained from the operating system.

- `setlocale( LC_ALL, "<localename>" );`

  Sets the locale to the locale name that's indicated by <*localename*>.

- `setlocale( LC_ALL, "<language>_<country>" );`

Sets the locale to the language and country/region indicated by *<language>* and *<country>*, together with the default code page obtained from the host operating system.

- `setlocale( LC_ALL, "<language>_<country>.<code_page>" );`

  Sets the locale to the language, country/region, and code page indicated by the *<language>*, *<country>*, and *<code_page>* strings. You can use various combinations of language, country/region, and code page. For example, this call sets the locale to French Canada with code page 1252:

  ```
  setlocale( LC_ALL, "French_Canada.1252" );
  ```

  This call sets the locale to French Canada with the default ANSI code page:

  ```
  setlocale( LC_ALL, "French_Canada.ACP" );
  ```

  This call sets the locale to French Canada with the default OEM code page:

  ```
  setlocale( LC_ALL, "French_Canada.OCP" );
  ```

- `setlocale( LC_ALL, "<language>" );`

  Sets the locale to the language that's indicated by *<language>*, and uses the default country/region for the specified language and the user-default ANSI code page for that country/region as obtained from the host operating system. For example, the following calls to **setlocale** are functionally equivalent:

  ```
  setlocale( LC_ALL, "en-US" );
  ```

  ```
  setlocale( LC_ALL, "English" );
  ```

  ```
  setlocale( LC_ALL, "English_United States.1252" );
  ```

  We recommend the first form for performance and maintainability.

- `setlocale( LC_ALL, ".<code_page>" );`

  Sets the code page to the value indicated by *<code_page>*, together with the default country/region and language (as defined by the host operating system) for the specified code page.

The category must be either **LC_ALL** or **LC_CTYPE** to effect a change of code page. For example, if the default country/region and language of the host operating system are "United States" and "English," the following two calls to **setlocale** are functionally equivalent:

```
setlocale( LC_ALL, ".1252" );
```

```
setlocale( LC_ALL, "English_United States.1252");
```

For more information, see the setlocale pragma directive in the C/C++ Preprocessor Reference.

The function _configthreadlocale is used to control whether **setlocale** affects the locale of all threads in a program or only the locale of the calling thread.

Requirements

ROUTINE	REQUIRED HEADER
setlocale	<locale.h>
_wsetlocale	<locale.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_setlocale.c
//
// This program demonstrates the use of setlocale when
// using two independent threads.
//

#include <locale.h>
#include <process.h>
#include <windows.h>
#include <stdio.h>
#include <time.h>

#define BUFF_SIZE 100

// Retrieve the date in the current
// locale's format.
int get_date(unsigned char* str)
{
    __time64_t ltime;
    struct tm  thetime;

    // Retrieve the current time
    _time64(&ltime);
    _gmtime64_s(&thetime, &ltime);

    // Format the current time structure into a string
    // "%#x" is the long date representation in the
    // current locale
    if (!strftime((char *)str, BUFF_SIZE, "%#x",
                  (const struct tm *)&thetime))
    {
        printf("strftime failed!\n");
        return -1;
    }
    return 0;
}

// This thread sets its locale to the argument
// and prints the date.
uintptr_t __stdcall SecondThreadFunc( void* pArguments )
{
    unsigned char str[BUFF_SIZE];
    char * locale = (char *)pArguments;

    // Set the thread locale
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, locale));

    // Retrieve the date string from the helper function
    if (get_date(str) == 0)
    {
        printf("The date in %s locale is: '%s'\n", locale, str);
    }

     endthreadex( 0 );
```

```
    return 0;
}

// The main thread sets the locale to English
// and then spawns a second thread (above) and prints the date.
int main()
{
    HANDLE          hThread;
    unsigned        threadID;
    unsigned char   str[BUFF_SIZE];

    // Enable per-thread locale causes all subsequent locale
    // setting changes in this thread to only affect this thread.
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);

    // Set the locale of the main thread to US English.
    printf("The thread locale is now set to %s.\n",
           setlocale(LC_ALL, "en-US"));

    // Create the second thread with a German locale.
    // Our thread function takes an argument of the locale to use.
    hThread = (HANDLE)_beginthreadex( NULL, 0, &SecondThreadFunc,
                                      "de-DE", 0, &threadID );

    if (get_date(str) == 0)
    {
        // Retrieve the date string from the helper function
        printf("The date in en-US locale is: '%s'\n\n", str);
    }

    // Wait for the created thread to finish.
    WaitForSingleObject( hThread, INFINITE );

    // Destroy the thread object.
    CloseHandle( hThread );
}
```

```
The thread locale is now set to en-US.
The time in en-US locale is: 'Wednesday, May 12, 2004'

The thread locale is now set to de-DE.
The time in de-DE locale is: 'Mittwoch, 12. Mai 2004'
```

See also

Locale Names, Languages, and Country/Region Strings
_configthreadlocale
_create_locale, _wcreate_locale
Locale
localeconv
_mbclen, mblen, _mblen_l
strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
_setmbcp
strcoll Functions
strftime, wcsftime, _strftime_l, _wcsftime_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l
wcstombs, _wcstombs_l
wctomb, _wctomb_l

_setmaxstdio

5/23/2019 • 2 minutes to read • Edit Online

Sets a maximum for the number of simultaneously open files at the stream I/O level.

Syntax

```
int _setmaxstdio(
   int new_max
);
```

Parameters

new_max
New maximum for the number of simultaneously open files at the stream I/O level.

Return Value

Returns *new_max* if successful; -1 otherwise.

If *new_max* is less than **_IOB_ENTRIES**, or greater than the maximum number of handles available in the operating system, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_setmaxstdio** function changes the maximum value for the number of files that may be open simultaneously at the stream I/O level.

C run-time I/O now supports up to 8,192 files open simultaneously at the low I/O level. This level includes files opened and accessed using the **_open**, **_read**, and **_write** family of I/O functions. By default, up to 512 files can be open simultaneously at the stream I/O level. This level includes files opened and accessed using the **fopen**, **fgetc**, and **fputc** family of functions. The limit of 512 open files at the stream I/O level can be increased to a maximum of 8,192 by use of the **_setmaxstdio** function.

Because stream I/O-level functions, such as **fopen**, are built on top of the low I/O-level functions, the maximum of 8,192 is a hard upper limit for the number of simultaneously open files accessed through the C run-time library.

> **NOTE**
> This upper limit might be beyond what's supported by a particular Win32 platform and configuration.

Requirements

ROUTINE	REQUIRED HEADER
_setmaxstdio	<stdio.h>

For more compatibility information, see Compatibility.

Example

See _getmaxstdio for an example of using **_setmaxstdio**.

See also

Stream I/O

_setmbcp

10/31/2018 • 2 minutes to read • Edit Online

Sets a new multibyte code page.

Syntax

```
int _setmbcp(
   int codepage
);
```

Parameters

codepage
New code page setting for locale-independent multibyte routines.

Return Value

Returns 0 if the code page is set successfully. If an invalid code page value is supplied for *codepage*, returns -1 and the code page setting is unchanged. Sets **errno** to **EINVAL** if a memory allocation failure occurs.

Remarks

The **_setmbcp** function specifies a new multibyte code page. By default, the run-time system automatically sets the multibyte code page to the system-default ANSI code page. The multibyte code page setting affects all multibyte routines that are not locale dependent. However, it is possible to instruct **_setmbcp** to use the code page defined for the current locale (see the following list of manifest constants and associated behavior results). For a list of the multibyte routines that are dependent on the locale code page rather than the multibyte code page, see Interpretation of Multibyte-Character Sequences.

The multibyte code page also affects multibyte-character processing by the following run-time library routines:

_exec functions	_mktemp	_stat
_fullpath	_spawn functions	_tempnam
_makepath	_splitpath	tmpnam

In addition, all run-time library routines that receive multibyte-character *argv* or *envp* program arguments as parameters (such as the **_exec** and **_spawn** families) process these strings according to the multibyte code page. Therefore, these routines are also affected by a call to **_setmbcp** that changes the multibyte code page.

The *codepage* argument can be set to any of the following values:

- **_MB_CP_ANSI** Use ANSI code page obtained from operating system at program startup.
- **_MB_CP_LOCALE** Use the current locale's code page obtained from a previous call to setlocale.
- **_MB_CP_OEM** Use OEM code page obtained from operating system at program startup.

- **_MB_CP_SBCS** Use single-byte code page. When the code page is set to **_MB_CP_SBCS**, a routine such as _ismbblead always returns false.
- Any other valid code page value, regardless of whether the value is an ANSI, OEM, or other operating-system-supported code page (except UTF-7 and UTF-8, which are not supported).

Requirements

ROUTINE	REQUIRED HEADER
_setmbcp	<mbctype.h>

For more compatibility information, see Compatibility.

See also

_getmbcp
setlocale, _wsetlocale

setmode

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _setmode instead.

_setmode

1/4/2019 • 2 minutes to read • Edit Online

Sets the file translation mode.

Syntax

```
int _setmode (
   int fd,
   int mode
);
```

Parameters

fd
File descriptor.

mode
New translation mode.

Return Value

If successful, returns the previous translation mode.

If invalid parameters are passed to this function, the invalid-parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to either **EBADF**, which indicates an invalid file descriptor, or **EINVAL**, which indicates an invalid *mode* argument.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_setmode** function sets to *mode* the translation mode of the file given by *fd*. Passing **_O_TEXT** as *mode* sets text (that is, translated) mode. Carriage return-line feed (CR-LF) combinations are translated into a single line feed character on input. Line feed characters are translated into CR-LF combinations on output. Passing **_O_BINARY** sets binary (untranslated) mode, in which these translations are suppressed.

You can also pass **_O_U16TEXT**, **_O_U8TEXT**, or **_O_WTEXT** to enable Unicode mode, as demonstrated in the second example later in this document.

Caution
Unicode mode is for wide print functions (for example, `wprintf`) and is not supported for narrow print functions. Use of a narrow print function on a Unicode mode stream triggers an assert.

_setmode is typically used to modify the default translation mode of **stdin** and **stdout**, but you can use it on any file. If you apply **_setmode** to the file descriptor for a stream, call **_setmode** before you perform any input or output operations on the stream.

Caution
If you write data to a file stream, explicitly flush the code by using fflush before you use **_setmode** to change the mode. If you do not flush the code, you might get unexpected behavior. If you have not written data to the stream, you do not have to flush the code.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_setmode	<io.h>	<fcntl.h>

For more compatibility information, see Compatibility.

Example

```
// crt_setmode.c
// This program uses _setmode to change
// stdin from text mode to binary mode.

#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main( void )
{
   int result;

   // Set "stdin" to have binary mode:
   result = _setmode( _fileno( stdin ), _O_BINARY );
   if( result == -1 )
      perror( "Cannot set mode" );
   else
      printf( "'stdin' successfully changed to binary mode\n" );
}
```

```
'stdin' successfully changed to binary mode
```

Example

```
// crt_setmodeunicode.c
// This program uses _setmode to change
// stdout to Unicode. Cyrillic and Ideographic
// characters will appear on the console (if
// your console font supports those character sets).

#include <fcntl.h>
#include <io.h>
#include <stdio.h>

int main(void) {
    _setmode(_fileno(stdout), _O_U16TEXT);
    wprintf(L"\x043a\x043e\x0448\x043a\x0430 \x65e5\x672c\x56fd\n");
    return 0;
}
```

See also

File Handling
_creat, _wcreat
fopen, _wfopen
_open, _wopen
_set_fmode

_set_new_handler

10/31/2018 • 3 minutes to read • Edit Online

Transfers control to your error-handling mechanism if the **new** operator fails to allocate memory.

Syntax

```
_PNH _set_new_handler( _PNH pNewHandler );
```

Parameters

pNewHandler
Pointer to the application-supplied memory handling function. An argument of 0 causes the new handler to be removed.

Return Value

Returns a pointer to the previous exception handling function registered by **_set_new_handler**, so that the previous function can be restored later. If no previous function has been set, the return value can be used to restore the default behavior; this value can be **NULL**.

Remarks

The C++ **_set_new_handler** function specifies an exception-handling function that gains control if the **new** operator fails to allocate memory. If **new** fails, the run-time system automatically calls the exception-handling function that was passed as an argument to **_set_new_handler**. **_PNH**, defined in New.h, is a pointer to a function that returns type **int** and takes an argument of type **size_t**. Use **size_t** to specify the amount of space to be allocated.

There is no default handler.

_set_new_handler is essentially a garbage-collection scheme. The run-time system retries allocation each time your function returns a nonzero value and fails if your function returns 0.

An occurrence of the **_set_new_handler** function in a program registers the exception-handling function specified in the argument list with the run-time system:

```
// set_new_handler1.cpp
#include <new.h>

int handle_program_memory_depletion( size_t )
{
   // Your code
}

int main( void )
{
   _set_new_handler( handle_program_memory_depletion );
   int *pi = new int[BIG_NUMBER];
}
```

You can save the function address that was last passed to the **_set_new_handler** function and reinstate it later:

```
_PNH old_handler = _set_new_handler( my_handler );
// Code that requires my_handler
// . . .
_set_new_handler( old_handler )
// Code that requires old_handler
// . . .
```

The C++ _set_new_mode function sets the new handler mode for malloc. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by **_set_new_handler**. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. To override the default, call:

```
_set_new_mode(1);
```

early in your program or link with Newmode.obj.

If a user-defined `operator new` is provided, the new handler functions are not automatically called on failure.

For more information, see new and delete in the *C++ Language Reference*.

There is a single **_set_new_handler** handler for all dynamically linked DLLs or executables; even if you call **_set_new_handler** your handler might be replaced by another or that you are replacing a handler set by another DLL or executable.

Requirements

ROUTINE	REQUIRED HEADER
_set_new_handler	<new.h>

For more compatibility information, see Compatibility.

Example

In this example, when the allocation fails, control is transferred to MyNewHandler. The argument passed to MyNewHandler is the number of bytes requested. The value returned from MyNewHandler is a flag indicating whether allocation should be retried: a nonzero value indicates that allocation should be retried, and a zero value indicates that allocation has failed.

```
// crt_set_new_handler.cpp
// compile with: /c
#include <stdio.h>
#include <new.h>
#define BIG_NUMBER 0x1fffffff

int coalesced = 0;

int CoalesceHeap()
{
   coalesced = 1;  // Flag RecurseAlloc to stop
   // do some work to free memory
   return 0;
}
// Define a function to be called if new fails to allocate memory.
int MyNewHandler( size_t size )
{
   printf("Allocation failed. Coalescing heap.\n");

   // Call a function to recover some heap space.
   return CoalesceHeap();
}

int RecurseAlloc() {
   int *pi = new int[BIG_NUMBER];
   if (!coalesced)
      RecurseAlloc();
   return 0;
}

int main()
{
   // Set the failure handler for new to be MyNewHandler.
   _set_new_handler( MyNewHandler );
   RecurseAlloc();
}
```

```
Allocation failed. Coalescing heap.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

See also

Memory Allocation
calloc
free
realloc

_set_new_mode

10/31/2018 • 2 minutes to read • Edit Online

Sets a new handler mode for **malloc**.

Syntax

```
int _set_new_mode( int newhandlermode );
```

Parameters

newhandlermode
New handler mode for **malloc**; valid value is 0 or 1.

Return Value

Returns the previous handler mode set for **malloc**. A return value of 1 indicates that, on failure to allocate memory, **malloc** previously called the new handler routine; a return value of 0 indicates that it did not. If the *newhandlermode* argument does not equal 0 or 1, returns -1.

Remarks

The C++ **_set_new_mode** function sets the new handler mode for malloc. The new handler mode indicates whether, on failure, **malloc** is to call the new handler routine as set by _set_new_handler. By default, **malloc** does not call the new handler routine on failure to allocate memory. You can override this default behavior so that, when **malloc** fails to allocate memory, **malloc** calls the new handler routine in the same way that the **new** operator does when it fails for the same reason. For more information, see the new and delete operators in the *C++ Language Reference*. To override the default, call:

```
_set_new_mode(1);
```

early in your program or link with Newmode.obj (see Link Options).

This function validates its parameter. If *newhandlermode* is anything other than 0 or 1, the function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **_set_new_mode** returns -1 and sets **errno** to `EINVAL`.

Requirements

ROUTINE	REQUIRED HEADER
_set_new_mode	<new.h>

For more compatibility information, see Compatibility.

See also

Memory Allocation
calloc

free
realloc
_query_new_handler
_query_new_mode

_set_printf_count_output

10/31/2018 • 2 minutes to read • Edit Online

Enable or disable support of the **%n** format in printf, _printf_l, wprintf, _wprintf_l-family functions.

Syntax

```
int _set_printf_count_output(
   int enable
);
```

Parameters

enable
A non-zero value to enable **%n** support, 0 to disable **%n** support.

Property Value/Return Value

The state of **%n** support before calling this function: non-zero if **%n** support was enabled, 0 if it was disabled.

Remarks

Because of security reasons, support for the **%n** format specifier is disabled by default in **printf** and all its variants. If **%n** is encountered in a **printf** format specification, the default behavior is to invoke the invalid parameter handler as described in Parameter Validation. Calling **_set_printf_count_output** with a non-zero argument will cause **printf**-family functions to interpret **%n** as described in Format Specification Syntax: printf and wprintf Functions.

Requirements

ROUTINE	REQUIRED HEADER
_set_printf_count_output	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_set_printf_count_output.c
#include <stdio.h>

int main()
{
   int e;
   int i;
   e = _set_printf_count_output( 1 );
   printf( "%%n support was %sabled.\n",
        e ? "en" : "dis" );
   printf( "%%n support is now %sabled.\n",
        _get_printf_count_output() ? "en" : "dis" );
   printf( "12345%n6789\n", &i ); // %n format should set i to 5
   printf( "i = %d\n", i );
}
```

```
%n support was disabled.
%n support is now enabled.
123456789
i = 5
```

See also

_get_printf_count_output

_set_se_translator

3/12/2019 • 4 minutes to read • Edit Online

Set a per-thread callback function to translate Win32 exceptions (C structured exceptions) into C++ typed exceptions.

Syntax

```
_se_translator_function _set_se_translator(
    _se_translator_function seTransFunction
);
```

Parameters

seTransFunction
Pointer to a C structured exception translator function that you write.

Return Value

Returns a pointer to the previous translator function registered by **_set_se_translator**, so that the previous function can be restored later. If no previous function has been set, the return value can be used to restore the default behavior; this value can be **nullptr**.

Remarks

The **_set_se_translator** function provides a way to handle Win32 exceptions (C structured exceptions) as C++ typed exceptions. To allow each C exception to be handled by a C++ **catch** handler, first define a C exception wrapper class that can be used, or derived from, to attribute a specific class type to a C exception. To use this class, install a custom C exception translator function that is called by the internal exception-handling mechanism each time a C exception is raised. Within your translator function, you can throw any typed exception that can be caught by a matching C++ **catch** handler.

You must use /EHa when using **_set_se_translator**.

To specify a custom translation function, call **_set_se_translator** using the name of your translation function as its argument. The translator function that you write is called once for each function invocation on the stack that has **try** blocks. There is no default translator function.

Your translator function should do no more than throw a C++ typed exception. If it does anything in addition to throwing (such as writing to a log file, for example) your program might not behave as expected, because the number of times the translator function is invoked is platform-dependent.

In a multithreaded environment, translator functions are maintained separately for each thread. Each new thread needs to install its own translator function. Thus, each thread is in charge of its own translation handling. **_set_se_translator** is specific to one thread; another DLL can install a different translation function.

The *seTransFunction* function that you write must be a native-compiled function (not compiled with /clr). It must take an unsigned integer and a pointer to a Win32 **_EXCEPTION_POINTERS** structure as arguments. The arguments are the return values of calls to the Win32 API **GetExceptionCode** and **GetExceptionInformation** functions, respectively.

```
typedef void (__cdecl *_se_translator_function)(unsigned int, struct _EXCEPTION_POINTERS* );
```

For **_set_se_translator**, there are implications when dynamically linking to the CRT; another DLL in the process might call **_set_se_translator** and replace your handler with its own.

When using **_set_se_translator** from managed code (code compiled with /clr) or mixed native and managed code, be aware that the translator affects exceptions generated in native code only. Any managed exceptions generated in managed code (such as when raising `System::Exception`) are not routed through the translator function. Exceptions raised in managed code using the Win32 function **RaiseException** or caused by a system exception like a divide by zero exception are routed through the translator.

Requirements

ROUTINE	REQUIRED HEADER
_set_se_translator	<eh.h>

For more compatibility information, see Compatibility.

Example

```
// crt_settrans.cpp
// compile with: cl /W4 /EHa crt_settrans.cpp
#include <stdio.h>
#include <windows.h>
#include <eh.h>
#include <exception>

class SE_Exception : public std::exception
{
private:
    unsigned int nSE;
public:
    SE_Exception() : nSE{ 0 } {}
    SE_Exception( unsigned int n ) : nSE{ n } {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc()
{
    __try
    {
        printf( "In __try, about to force exception\n" );
        int x = 5;
        int y = 0;
        int *p = &y;
        *p = x / *p;
    }
    __finally
    {
        printf( "In __finally\n" );
    }
}

void trans_func(unsigned int u, EXCEPTION_POINTERS*)
{
    throw SE_Exception(u);
}

int main()
{
    auto original = _set_se_translator(trans_func);
    try
    {
        SEFunc();
    }
    catch(SE_Exception& e)
    {
        printf("Caught a __try exception, error %8.8x.\n", e.getSeNumber());
    }
    _set_se_translator(original);
}
```

```
In __try, about to force exception
In __finally
Caught a __try exception, error c0000094.
```

Example

Although the functionality provided by **_set_se_translator** is not available in managed code, it is possible to use this mapping in native code, even if that native code is in a compilation under the **/clr** switch, as long as the native code is indicated using `#pragma unmanaged`. If a structured exception is being thrown in managed code that is to be mapped, the code that generates and handles the exception must be marked `#pragma unmanaged`. The

following code shows a possible use. For more information, see Pragma Directives and the __Pragma Keyword.

```
// crt_set_se_translator_clr.cpp
// compile with: cl /W4 /clr crt_set_se_translator_clr.cpp
#include <windows.h>
#include <eh.h>
#include <assert.h>
#include <stdio.h>
#include <exception>

int thrower_func(int i) {
   int y = 0;
   int *p = &y;
   *p = i / *p;
   return 0;
}

class SE_Exception : public std::exception {
private:
    unsigned int nSE;
public:
    SE_Exception() : nSE{ 0 } {}
    SE_Exception(unsigned int n) : nSE{ n } {}
    unsigned int getSeNumber() { return nSE; }
};

#pragma unmanaged
void my_trans_func(unsigned int u, PEXCEPTION_POINTERS)
{
    throw SE_Exception(u);
}

void DoTest()
{
    try
    {
        thrower_func(10);
    }
    catch(SE_Exception& e)
    {
        printf("Caught SE_Exception, error %8.8x\n", e.getSeNumber());
    }
    catch(...)
    {
        printf("Caught unexpected SEH exception.\n");
    }
}
#pragma managed

int main() {
    auto original = _set_se_translator(my_trans_func);
    DoTest();
    _set_se_translator(original);
}
```

```
Caught SE_Exception, error c0000094
```

See also

Exception Handling Routines
set_terminate
set_unexpected
terminate

unexpected

_set_SSE2_enable

10/31/2018 • 2 minutes to read • Edit Online

Enables or disables the use of Streaming SIMD Extensions 2 (SSE2) instructions in CRT math routines. (This function is not available on x64 architectures because SSE2 is enabled by default.)

Syntax

```
int _set_SSE2_enable(
   int flag
);
```

Parameters

flag
1 to enable the SSE2 implementation; 0 to disable the SSE2 implementation. By default, SSE2 implementation is enabled on processors that support it.

Return Value

Nonzero if SSE2 implementation is enabled; zero if SSE2 implementation is disabled.

Remarks

The following functions have SSE2 implementations that can be enabled by using **_set_SSE2_enable**:

- atan
- ceil
- exp
- floor
- log
- log10
- modf
- pow

The SSE2 implementations of these functions might give slightly different answers than the default implementations, because SSE2 intermediate values are 64-bit floating-point quantities but the default implementation intermediate values are 80-bit floating-point quantities.

NOTE

If you use the /Oi (Generate Intrinsic Functions) compiler option to compile the project, it may appear that **_set_SSE2_enable** has no effect. The **/Oi** compiler option gives the compiler the authority to use intrinsics to replace CRT calls; this behavior overrides the effect of **_set_SSE2_enable**. If you want to guarantee that **/Oi** does not override **_set_SSE2_enable**, use **/Oi-** to compile your project. This might also be good practice when you use other compiler switches that imply **/Oi**.

The SSE2 implementation is only used if all exceptions are masked. Use _control87, _controlfp to mask exceptions.

Requirements

ROUTINE	REQUIRED HEADER
_set_SSE2_enable	<math.h>

For more compatibility information, see Compatibility.

Example

```
// crt_set_SSE2_enable.c
// processor: x86
#include <math.h>
#include <stdio.h>

int main()
{
   int i = _set_SSE2_enable(1);

   if (i)
      printf("SSE2 enabled.\n");
   else
      printf("SSE2 not enabled; processor does not support SSE2.\n");
}
```

```
SSE2 enabled.
```

See also

CRT Library Features

set_terminate (CRT)

10/31/2018 • 2 minutes to read • Edit Online

Installs your own termination routine to be called by **terminate**.

Syntax

```
terminate_function set_terminate( terminate_function termFunction );
```

Parameters

termFunction
Pointer to a terminate function that you write.

Return Value

Returns a pointer to the previous function registered by **set_terminate** so that the previous function can be restored later. If no previous function has been set, the return value may be used to restore the default behavior; this value may be **NULL**.

Remarks

The **set_terminate** function installs *termFunction* as the function called by **terminate**. **set_terminate** is used with C++ exception handling and may be called at any point in your program before the exception is thrown. **terminate** calls abort by default. You can change this default by writing your own termination function and calling **set_terminate** with the name of your function as its argument. **terminate** calls the last function given as an argument to **set_terminate**. After performing any desired cleanup tasks, *termFunction* should exit the program. If it does not exit (if it returns to its caller), abort is called.

In a multithreaded environment, terminate functions are maintained separately for each thread. Each new thread needs to install its own terminate function. Thus, each thread is in charge of its own termination handling.

The **terminate_function** type is defined in EH.H as a pointer to a user-defined termination function, *termFunction* that returns **void**. Your custom function *termFunction* can take no arguments and should not return to its caller. If it does, abort is called. An exception may not be thrown from within *termFunction*.

```
typedef void ( *terminate_function )( );
```

> **NOTE**
> The **set_terminate** function only works outside the debugger.

There is a single **set_terminate** handler for all dynamically linked DLLs or EXEs; even if you call **set_terminate** your handler may be replaced by another, or you may be replacing a handler set by another DLL or EXE.

Requirements

ROUTINE	REQUIRED HEADER
set_terminate	<eh.h>

For additional compatibility information, see Compatibility.

Example

See the example for terminate.

See also

Exception Handling Routines
abort
_get_terminate
set_unexpected
terminate
unexpected

set_unexpected (CRT)

10/31/2018 • 2 minutes to read • Edit Online

Installs your own termination function to be called by **unexpected**.

Syntax

```
unexpected_function set_unexpected( unexpected_function unexpFunction );
```

Parameters

unexpFunction
Pointer to a function that you write to replace the **unexpected** function.

Return Value

Returns a pointer to the previous termination function registered by **_set_unexpected** so that the previous function can be restored later. If no previous function has been set, the return value may be used to restore the default behavior; this value may be **NULL**.

Remarks

The **set_unexpected** function installs *unexpFunction* as the function called by **unexpected**. **unexpected** is not used in the current C++ exception-handling implementation. The **unexpected_function** type is defined in EH.H as a pointer to a user-defined unexpected function, *unexpFunction* that returns **void**. Your custom *unexpFunction* function should not return to its caller.

```
typedef void ( *unexpected_function )( );
```

By default, **unexpected** calls **terminate**. You can change this default behavior by writing your own termination function and calling **set_unexpected** with the name of your function as its argument. **unexpected** calls the last function given as an argument to **set_unexpected**.

Unlike the custom termination function installed by a call to **set_terminate**, an exception can be thrown from within *unexpFunction*.

In a multithreaded environment, unexpected functions are maintained separately for each thread. Each new thread needs to install its own unexpected function. Thus, each thread is in charge of its own unexpected handling.

In the current Microsoft implementation of C++ exception handling, **unexpected** calls **terminate** by default and is never called by the exception-handling run-time library. There is no particular advantage to calling **unexpected** rather than **terminate**.

There is a single **set_unexpected** handler for all dynamically linked DLLs or EXEs; even if you call **set_unexpected** your handler may be replaced by another or that you are replacing a handler set by another DLL or EXE.

Requirements

ROUTINE	REQUIRED HEADER
set_unexpected	<eh.h>

For additional compatibility information, see Compatibility.

See also

Exception Handling Routines
abort
_get_unexpected
set_terminate
terminate
unexpected

setvbuf

11/8/2018 • 2 minutes to read • Edit Online

Controls stream buffering and buffer size.

Syntax

```
int setvbuf(
   FILE *stream,
   char *buffer,
   int mode,
   size_t size
);
```

Parameters

stream
Pointer to **FILE** structure.

buffer
User-allocated buffer.

mode
Mode of buffering.

size
Buffer size in bytes. Allowable range: 2 <= *size* <= INT_MAX (2147483647). Internally, the value supplied for *size* is rounded down to the nearest multiple of 2.

Return Value

Returns 0 if successful.

If *stream* is **NULL**, or if *mode* or *size* is not within a valid change, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns -1 and sets **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **setvbuf** function allows the program to control both buffering and buffer size for *stream*. *stream* must refer to an open file that has not undergone an I/O operation since it was opened. The array pointed to by *buffer* is used as the buffer, unless it is **NULL**, in which case **setvbuf** uses an automatically allocated buffer of length *size*/2 * 2 bytes.

The mode must be **_IOFBF**, **_IOLBF**, or **_IONBF**. If *mode* is **_IOFBF** or **_IOLBF**, then *size* is used as the size of the buffer. If *mode* is **_IONBF**, the stream is unbuffered and *size* and *buffer* are ignored. Values for *mode* and their meanings are:

MODE VALUE	MEANING
_IOFBF	Full buffering; that is, *buffer* is used as the buffer and *size* is used as the size of the buffer. If *buffer* is **NULL**, an automatically allocated buffer *size* bytes long is used.
_IOLBF	For some systems, this provides line buffering. However, for Win32, the behavior is the same as **_IOFBF** - Full Buffering.
_IONBF	No buffer is used, regardless of *buffer* or *size*.

Requirements

ROUTINE	REQUIRED HEADER
setvbuf	<stdio.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_setvbuf.c
// This program opens two streams: stream1
// and stream2. It then uses setvbuf to give stream1 a
// user-defined buffer of 1024 bytes and stream2 no buffer.
//

#include <stdio.h>

int main( void )
{
   char buf[1024];
   FILE *stream1, *stream2;

   if( fopen_s( &stream1, "data1", "a" ) == 0 &&
       fopen_s( &stream2, "data2", "w" ) == 0 )
   {
      if( setvbuf( stream1, buf, _IOFBF, sizeof( buf ) ) != 0 )
         printf( "Incorrect type or size of buffer for stream1\n" );
      else
         printf( "'stream1' now has a buffer of 1024 bytes\n" );
      if( setvbuf( stream2, NULL, _IONBF, 0 ) != 0 )
         printf( "Incorrect type or size of buffer for stream2\n" );
      else
         printf( "'stream2' now has no buffer\n" );
      _fcloseall();
   }
}
```

```
'stream1' now has a buffer of 1024 bytes
'stream2' now has no buffer
```

See also

Stream I/O
fclose, _fcloseall
fflush
fopen, _wfopen
setbuf

signal

11/8/2018 • 5 minutes to read • Edit Online

Sets interrupt signal handling.

> **IMPORTANT**
> Do not use this method to shut down a Microsoft Store app, except in testing or debugging scenarios. Programmatic or UI ways to close a Store app are not permitted according to the Microsoft Store policies. For more information, see UWP app lifecycle.

Syntax

```
void __cdecl *signal(int sig, int (*func)(int, int));
```

Parameters

sig
Signal value.

func
The second parameter is a pointer to the function to be executed. The first parameter is a signal value and the second parameter is a sub-code that can be used when the first parameter is SIGFPE.

Return Value

signal returns the previous value of func that's associated with the given signal. For example, if the previous value of *func* was **SIG_IGN**, the return value is also **SIG_IGN**. A return value of **SIG_ERR** indicates an error; in that case, **errno** is set to **EINVAL**.

See errno, _doserrno, _sys_errlist, and _sys_nerr for more information about return codes.

Remarks

The **signal** function enables a process to choose one of several ways to handle an interrupt signal from the operating system. The *sig* argument is the interrupt to which **signal** responds; it must be one of the following manifest constants, which are defined in SIGNAL.H.

SIG VALUE	DESCRIPTION
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access

SIG VALUE	DESCRIPTION
SIGTERM	Termination request

If *sig* is not one of the above values, the invalid parameter handler is invoked, as defined in Parameter Validation. If execution is allowed to continue, this function sets **errno** to **EINVAL** and returns **SIG_ERR**.

By default, **signal** terminates the calling program with exit code 3, regardless of the value of *sig*.

> **NOTE**
>
> **SIGINT** is not supported for any Win32 application. When a CTRL+C interrupt occurs, Win32 operating systems generate a new thread to specifically handle that interrupt. This can cause a single-thread application, such as one in UNIX, to become multithreaded and cause unexpected behavior.

The *func* argument is an address to a signal handler that you write, or to one of the predefined constants **SIG_DFL** or **SIG_IGN**, which are also defined in SIGNAL.H. If *func* is a function, it is installed as the signal handler for the given signal. The signal handler's prototype requires one formal argument, *sig*, of type **int**. The operating system provides the actual argument through *sig* when an interrupt occurs; the argument is the signal that generated the interrupt. Therefore, you can use the six manifest constants (listed in the preceding table) in your signal handler to determine which interrupt occurred and take appropriate action. For example, you can call **signal** twice to assign the same handler to two different signals, and then test the *sig* argument in the handler to take different actions based on the signal received.

If you are testing for floating-point exceptions (**SIGFPE**), *func* points to a function that takes an optional second argument that is one of several manifest constants, defined in FLOAT.H, of the form **FPE_xxx**. When a **SIGFPE** signal occurs, you can test the value of the second argument to determine the kind of floating-point exception and then take appropriate action. This argument and its possible values are Microsoft extensions.

For floating-point exceptions, the value of *func* is not reset when the signal is received. To recover from floating-point exceptions, use try/except clauses to surround the floating point operations. It's also possible to recover by using setjmp with longjmp. In either case, the calling process resumes execution and leaves the floating-point state of the process undefined.

If the signal handler returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the kind of signal or operating mode.

Before the specified function is executed, the value of *func* is set to **SIG_DFL**. The next interrupt signal is treated as described for **SIG_DFL**, unless an intervening call to **signal** specifies otherwise. You can use this feature to reset signals in the called function.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, your signal-handler function may get control when a run-time operation is incomplete and in an unknown state. The following list summarizes the restrictions that determine which functions you can use in your signal-handler routine.

- Do not issue low-level or STDIO.H I/O routines (for example, **printf** or **fread**).
- Do not call heap routines or any routine that uses the heap routines (for example, **malloc**, **_strdup**, or **_putenv**). See malloc for more information.
- Do not use any function that generates a system call (for example, **_getcwd** or **time**).
- Do not use **longjmp** unless the interrupt is caused by a floating-point exception (that is, *sig* is **SIGFPE**). In this case, first reinitialize the floating-point package by using a call to **_fpreset**.
- Do not use any overlay routines.

A program must contain floating-point code if it is to trap the **SIGFPE** exception by using the function. If your program does not have floating-point code and requires the run-time library's signal-handling code, just declare a volatile double and initialize it to zero:

```
volatile double d = 0.0f;
```

The **SIGILL** and **SIGTERM** signals are not generated under Windows. They are included for ANSI compatibility. Therefore, you can set signal handlers for these signals by using **signal**, and you can also explicitly generate these signals by calling raise.

Signal settings are not preserved in spawned processes that are created by calls to _exec or _spawn functions. The signal settings are reset to the default values in the new process.

Requirements

ROUTINE	REQUIRED HEADER
signal	<signal.h>

For additional compatibility information, see Compatibility.

Example

The following example shows how to use **signal** to add some custom behavior to the **SIGABRT** signal. For additional information about abort behavior, see _set_abort_behavior.

```
// crt_signal.c
// compile with: /EHsc /W4
// Use signal to attach a signal handler to the abort routine
#include <stdlib.h>
#include <signal.h>
#include <tchar.h>

void SignalHandler(int signal)
{
    if (signal == SIGABRT) {
        // abort signal handler code
    } else {
        // ...
    }
}

int main()
{
    typedef void (*SignalHandlerPointer)(int);

    SignalHandlerPointer previousHandler;
    previousHandler = signal(SIGABRT, SignalHandler);

    abort();
}
```

```
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

See also

signbit

2/4/2019 • 2 minutes to read • Edit Online

Determines whether a floating-point value is negative.

Syntax

```
int signbit(
   /* floating-point */ x
); /* C-only macro */

inline bool signbit(
   float x
) throw(); /* C++-only overloaded function */

inline bool signbit(
   double x
) throw(); /* C++-only overloaded function */

inline bool signbit(
   long double x
) throw(); /* C++-only overloaded function */
```

Parameters

x
The floating-point value to test.

Return value

signbit returns a non-zero value (**true** in C++) if the argument *x* is negative or negative infinity. It returns 0 (**false** in C++) if the argument is non-negative, positive infinity, or a NAN.

Remarks

signbit is a macro when compiled as C, and an overloaded inline function when compiled as C++.

Requirements

FUNCTION	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
signbit	<math.h>	<math.h> or <cmath>

For more compatibility information, see Compatibility.

See also

Floating-Point Support
isfinite, _finite, _finitef
isinf
isnan, _isnan, _isnanf
isnormal

_fpclass, _fpclassf

sin, sinf, sinl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the sine of a floating-point value.

Syntax

```
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

```
float sin(float x);  // C++ only
long double sin(long double x);  // C++ only
```

Parameters

x
Angle in radians.

Return value

The **sin** functions return the sine of *x*. If *x* is greater than or equal to 263, or less than or equal to -263, a loss of significance in the result occurs.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	None	_DOMAIN
± ∞ (sin, sinf, sinl)	INVALID	_DOMAIN

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Because C++ allows overloading, you can call overloads of **sin** that take and return **float** or **long double** values. In a C program, **sin** always takes and returns **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
sin, **sinf**, **sinl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sincos.c
// This program displays the sine and cosine of pi / 2.
// Compile by using: cl /W4 crt_sincos.c

#include <math.h>
#include <stdio.h>

int main( void)
{
   double pi = 3.1415926535;
   double x, y;

   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
}
```

```
sin( 1.570796 ) = 1.000000
cos( 1.570796 ) = 0.000000
```

See also

Floating-Point Support
acos, acosf, acosl
asin, asinf, asinl
atan, atanf, atanl, atan2, atan2f, atan2l
cos, cosf, cosl
tan, tanf, tanl
_CIsin

sinh, sinhf, sinhl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the hyperbolic sine.

Syntax

```
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

```
float sinh(float x);  // C++ only
long double sinh(long double x);  // C++ only
```

Parameters

x
Angle in radians.

Return Value

The **sinh** functions return the hyperbolic sine of *x*. By default, if the result is too large, **sinh** sets **errno** to **ERANGE** and returns ±**HUGE_VAL**.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	None	_DOMAIN
|x| ≥ 7.104760e+002	OVERFLOW+INEXACT	OVERFLOW

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Because C++ allows overloading, you can call overloads of **sinh** that take and return **float** or **long double** values. In a C program, **sinh** always takes and returns **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
sinh, **sinhf**, **sinhl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sinhcosh.c
// This program displays the hyperbolic
// sine and hyperbolic cosine of pi / 2.
// Compile by using: cl /W4 crt_sinhcosh.c

#include <math.h>
#include <stdio.h>

int main( void)
{
   double pi = 3.1415926535;
   double x, y;

   x = pi / 2;
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

```
sinh( 1.570796 ) = 2.301299
cosh( 1.570796 ) = 2.509178
```

See also

Floating-Point Support
acosh, acoshf, acoshl
asinh, asinhf, asinhl
atanh, atanhf, atanhl
cosh, coshf, coshl
tanh, tanhf, tanhl

snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l

10/31/2018 • 5 minutes to read • Edit Online

Writes formatted data to a string. More secure versions of these functions are available; see _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l.

Syntax

```
int snprintf(
   char *buffer,
   size_t count,
   const char *format [,
   argument] ...
);
int _snprintf(
   char *buffer,
   size_t count,
   const char *format [,
   argument] ...
);
int _snprintf_l(
   char *buffer,
   size_t count,
   const char *format,
   locale_t locale [,
   argument] ...
);
int _snwprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format [,
   argument] ...
);
int _snwprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
template <size_t size>
int _snprintf(
   char (&buffer)[size],
   size_t count,
   const char *format [,
   argument] ...
); // C++ only
template <size_t size>
int _snprintf_l(
   char (&buffer)[size],
   size_t count,
   const char *format,
   locale_t locale [,
   argument] ...
); // C++ only
template <size_t size>
int _snwprintf(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format [,
   argument] ...
); // C++ only
template <size_t size>
int _snwprintf_l(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   locale_t locale [,
   argument] ...
); // C++ only
```

Parameters

buffer

Storage location for the output.

count
Maximum number of characters to store.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

For more information, see Format Specification Syntax: printf and wprintf Functions.

Return Value

Let **len** be the length of the formatted data string, not including the terminating null. Both **len** and *count* are in bytes for **snprintf** and **_snprintf**, wide characters for **_snwprintf**.

For all functions, if **len** < *count*, **len** characters are stored in *buffer*, a null-terminator is appended, and **len** is returned.

The **snprintf** function truncates the output when **len** is greater than or equal to *count*, by placing a null-terminator at `buffer[count-1]`. The value returned is **len**, the number of characters that would have been output if *count* was large enough. The **snprintf** function returns a negative value if an encoding error occurs.

For all functions other than **snprintf**, if **len** = *count*, **len** characters are stored in *buffer*, no null-terminator is appended, and **len** is returned. If **len** > *count*, *count* characters are stored in *buffer*, no null-terminator is appended, and a negative value is returned.

If *buffer* is a null pointer and *count* is zero, **len** is returned as the count of characters required to format the output, not including the terminating null. To make a successful call with the same *argument* and *locale* parameters, allocate a buffer holding at least **len** + 1 characters.

If *buffer* is a null pointer and *count* is nonzero, or if *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **snprintf** function and the **_snprintf** family of functions format and store *count* or fewer characters in *buffer*. The **snprintf** function always stores a terminating null character, truncating the output if necessary. The **_snprintf** family of functions only appends a terminating null character if the formatted string length is strictly less than *count* characters. Each *argument* (if any) is converted and is output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. If copying occurs between strings that overlap, the behavior is undefined.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. Because the **_snprintf** functions do not guarantee null termination—in particular, when the return value is *count*—make sure that they are followed by code that adds the null terminator. For more information, see Avoiding Buffer Overruns.

Beginning with the UCRT in Visual Studio 2015 and Windows 10, **snprintf** is no longer identical to **_snprintf**. The **snprintf** function behavior is now C99 standard compliant.

_snwprintf is a wide-character version of **_snprintf**; the pointer arguments to **_snwprintf** are wide-character strings. Detection of encoding errors in **_snwprintf** might differ from that in **_snprintf**. **_snwprintf**, just like **swprintf**, writes output to a string instead of a destination of type **FILE**.

The versions of these functions that have the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, these functions have template overloads that invoke their newer, more secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sntprintf	**_snprintf**	**_snprintf**	**_snwprintf**
_sntprintf_l	**_snprintf_l**	**_snprintf_l**	**_snwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
snprintf, **_snprintf**, **_snprintf_l**	<stdio.h>
_snwprintf, **_snwprintf_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_snprintf.c
// compile with: /W3
#include <stdio.h>
#include <stdlib.h>

#if !defined(__cplusplus)
typedef int bool;
const bool true = 1;
const bool false = 0;
#endif

#define FAIL 0 // change to 1 and see what happens

int main(void)
{
   char buffer[200];
   const static char s[] = "computer"
#if FAIL
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
"computercomputercomputercomputercomputercomputercomputercomputercomputer"
#endif
   ;
   const char c = 'l';
   const int i = 35;
```

```
#if FAIL
   const double fp = 1e300; // doesn't fit in the buffer
#else
   const double fp = 1.7320534;
#endif
   /* !subtract one to prevent "squeezing out" the terminal null! */
   const int bufferSize = sizeof(buffer)/sizeof(buffer[0]) - 1;
   int bufferUsed = 0;
   int bufferLeft = bufferSize - bufferUsed;
   bool bSuccess = true;
   buffer[0] = 0;

   /* Format and print various data: */

   if (bufferLeft > 0)
   {
      int perElementBufferUsed = _snprintf(&buffer[bufferUsed],
      bufferLeft, "   String: %s\n", s ); // C4996
      // Note: _snprintf is deprecated; consider _snprintf_s instead
      if (bSuccess = (perElementBufferUsed >= 0))
      {
         bufferUsed += perElementBufferUsed;
         bufferLeft -= perElementBufferUsed;
         if (bufferLeft > 0)
         {
            int perElementBufferUsed = _snprintf(&buffer[bufferUsed],
            bufferLeft, "   Character: %c\n", c ); // C4996
            if (bSuccess = (perElementBufferUsed >= 0))
            {
               bufferUsed += perElementBufferUsed;
               bufferLeft -= perElementBufferUsed;
               if (bufferLeft > 0)
               {
                  int perElementBufferUsed = _snprintf(&buffer
                  [bufferUsed], bufferLeft, "   Integer: %d\n", i ); // C4996
                  if (bSuccess = (perElementBufferUsed >= 0))
                  {
                     bufferUsed += perElementBufferUsed;
                     bufferLeft -= perElementBufferUsed;
                     if (bufferLeft > 0)
                     {
                        int perElementBufferUsed = _snprintf(&buffer
                        [bufferUsed], bufferLeft, "   Real: %f\n", fp ); // C4996
                        if (bSuccess = (perElementBufferUsed >= 0))
                        {
                           bufferUsed += perElementBufferUsed;
                        }
                     }
                  }
               }
            }
         }
      }
   }

   if (!bSuccess)
   {
      printf("%s\n", "failure");
   }
   else
   {
      /* !store null because _snprintf doesn't necessarily (if the string
       * fits without the terminal null, but not with it)!
       * bufferUsed might be as large as bufferSize, which normally is
       * like going one element beyond a buffer, but in this case
       * subtracted one from bufferSize, so we're ok.
       */
      buffer[bufferUsed] = 0;
      printf( "Output:\n%s\ncharacter count = %d\n", buffer, bufferUsed );
```

```
    }
    return EXIT_SUCCESS;
}
```

```
Output:
   String: computer
   Character: l
   Integer: 35
   Real: 1.732053

character count = 69
```

See also

Stream I/O
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l

3/1/2019 • 4 minutes to read • Edit Online

Writes formatted data to a string. These are versions of snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int _snprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format [,
   argument] ...
);
int _snprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   locale_t locale [,
   argument] ...
);
int _snwprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format [,
   argument] ...
);
int _snwprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
template <size_t size>
int _snprintf_s(
   char (&buffer)[size],
   size_t count,
   const char *format [,
   argument] ...
); // C++ only
template <size_t size>
int _snwprintf_s(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format [,
   argument] ...
); // C++ only
```

Parameters

buffer
Storage location for the output.

sizeOfBuffer

The size of the storage location for output. Size in **bytes** for **_snprintf_s** or size in **words** for **_snwprintf_s**.

count
Maximum number of characters to store, or _TRUNCATE.

format
Format-control string.

argument
Optional arguments.

locale
The locale to use.

Return Value

_snprintf_s returns the number of characters stored in *buffer*, not counting the terminating null character. **_snwprintf_s** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

If the storage required to store the data and a terminating null exceeds *sizeOfBuffer*, the invalid parameter handler is invoked, as described in Parameter Validation. If execution continues after the invalid parameter handler, these functions set *buffer* to an empty string, set **errno** to **ERANGE**, and return -1.

If *buffer* or *format* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_snprintf_s** function formats and stores *count* or fewer characters in *buffer* and appends a terminating null. Each argument (if any) is converted and output according to the corresponding format specification in *format*. The formatting is consistent with the **printf** family of functions; see Format Specification Syntax: printf and wprintf Functions. If copying occurs between strings that overlap, the behavior is undefined.

If *count* is _TRUNCATE, then **_snprintf_s** writes as much of the string as will fit in *buffer* while leaving room for a terminating null. If the entire string (with terminating null) fits in *buffer*, then **_snprintf_s** returns the number of characters written (not including the terminating null); otherwise, **_snprintf_s** returns -1 to indicate that truncation occurred.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string.

_snwprintf_s is a wide-character version of **_snprintf_s**; the pointer arguments to **_snwprintf_s** are wide-character strings. Detection of encoding errors in **_snwprintf_s** might differ from that in **_snprintf_s**. **_snwprintf_s**, like **swprintf_s**, writes output to a string rather than to a destination of type **FILE**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sntprintf_s	**_snprintf_s**	**_snprintf_s**	**_snwprintf_s**
_sntprintf_s_l	**_snprintf_s_l**	**_snprintf_s_l**	**_snwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_snprintf_s, **_snprintf_s_l**	<stdio.h>
_snwprintf_s, **_snwprintf_s_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_snprintf_s.cpp
// compile with: /MTd

// These #defines enable secure template overloads
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

int snprintf_s_tester( const char * fmt, int x, size_t count )
{
   char dest[10];

   printf( "\n" );

   if ( count == _TRUNCATE )
      printf( "%zd-byte buffer; truncation semantics\n",
                _countof(dest) );
   else
      printf( "count = %zd; %zd-byte buffer\n",
                count, _countof(dest) );

   int ret = _snprintf_s( dest, _countof(dest), count, fmt, x );

   printf( "    new contents of dest: '%s'\n", dest );

   return ret;
}

void Examples()
{
   // formatted output string is 9 characters long: "<<<123>>>"
   snprintf_s_tester( "<<<%d>>>", 121, 8 );
   snprintf_s_tester( "<<<%d>>>", 121, 9 );
   snprintf_s_tester( "<<<%d>>>", 121, 10 );
```

```
   printf( "\nDestination buffer too small:\n" );

   snprintf_s_tester( "<<<%d>>>", 1221, 10 );

   printf( "\nTruncation examples:\n" );

   int ret = snprintf_s_tester( "<<<%d>>>", 1221, _TRUNCATE );
   printf( "    truncation %s occur\n", ret == -1 ? "did"
                                                  : "did not" );

   ret = snprintf_s_tester( "<<<%d>>>", 121, _TRUNCATE );
   printf( "    truncation %s occur\n", ret == -1 ? "did"
                                                  : "did not" );
   printf( "\nSecure template overload example:\n" );

   char dest[10];
   _snprintf( dest, 10, "<<<%d>>>", 12321 );
   // With secure template overloads enabled (see #defines
   // at top of file), the preceding line is replaced by
   //    _snprintf_s( dest, _countof(dest), 10, "<<<%d>>>", 12345 );
   // Instead of causing a buffer overrun, _snprintf_s invokes
   // the invalid parameter handler.
   // If secure template overloads were disabled, _snprintf would
   // write 10 characters and overrun the dest buffer.
   printf( "    new contents of dest: '%s'\n", dest );
}

void myInvalidParameterHandler(
   const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf(L"Invalid parameter handler invoked: %s\n", expression);
}

int main( void )
{
   _invalid_parameter_handler oldHandler, newHandler;

   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);
   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   Examples();
}
```

```
count = 8; 10-byte buffer
    new contents of dest: '<<<121>>'

count = 9; 10-byte buffer
    new contents of dest: '<<<121>>>'

count = 10; 10-byte buffer
    new contents of dest: '<<<121>>>'

Destination buffer too small:

count = 10; 10-byte buffer
Invalid parameter handler invoked: ("Buffer too small", 0)
    new contents of dest: ''

Truncation examples:

10-byte buffer; truncation semantics
    new contents of dest: '<<<1221>>'
    truncation did occur

10-byte buffer; truncation semantics
    new contents of dest: '<<<121>>>'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: ("Buffer too small", 0)
    new contents of dest: ''
```

See also

Stream I/O
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_snscanf, _snscanf_l, _snwscanf, _snwscanf_l

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data of a specified length from a string. More secure versions of these functions are available; see _snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l.

Syntax

```
int __cdecl _snscanf(
   const char * input,
   size_t length,
   const char * format,
   ...
);
int __cdecl _snscanf_l(
   const char * input,
   size_t length,
   const char * format,
   locale_t locale,
   ...
);
int __cdecl _snwscanf(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   ...
);
int __cdecl _snwscanf_l(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   locale_t locale,
   ...
);
```

Parameters

input
Input string to examine.

length
Number of characters to examine in *input*.

format
One or more format specifiers.

...
Optional variables that will be used to store the values extracted from the input string by the format specifiers in *format*.

locale
The locale to use.

Return Value

Both of these functions returns the number of fields successfully converted and assigned; the return value does

not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion. For more information, see sscanf.

If *input* or *format* is a **NULL** pointer, or if *length* is less than or equal to zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

This function is like **sscanf** except that it provides the ability to specify a fixed number of characters to examine from the input string. For more information, see sscanf.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sntscanf	**_snscanf**	**_snscanf**	**_snwscanf**
_sntscanf_l	**_snscanf_l**	**_snscanf_l**	**_snwscanf_l**

Requirements

ROUTINE	REQUIRED HEADER
_snscanf, **_snscanf_l**	<stdio.h>
_snwscanf, **_snwscanf_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_snscanf.c
// compile with: /W3

#include <stdio.h>
int main( )
{
   char  str1[] = "15 12 14...";
   wchar_t  str2[] = L"15 12 14...";
   char  s1[3];
   wchar_t  s2[3];
   int   i;
   float fp;

   i = _snscanf( str1, 6,  "%s %f", s1, &fp); // C4996
   // Note: _snscanf is deprecated; consider using _snscanf_s instead
   printf("_snscanf converted %d fields: ", i);
   printf("%s and %f\n", s1, fp);

   i = _snwscanf( str2, 6,  L"%s %f", s2, &fp); // C4996
   // Note: _snwscanf is deprecated; consider using _snwscanf_s instead
   wprintf(L"_snwscanf converted %d fields: ", i);
   wprintf(L"%s and %f\n", s2, fp);
}
```

```
_snscanf converted 2 fields: 15 and 12.000000
_snwscanf converted 2 fields: 15 and 12.000000
```

See also

scanf Width Specification

_snscanf_s, _snscanf_s_l, _snwscanf_s, _snwscanf_s_l

3/1/2019 • 2 minutes to read • Edit Online

Reads formatted data of a specified length from a string. These are versions of _snscanf, _snscanf_l, _snwscanf, _snwscanf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int __cdecl _snscanf_s(
   const char * input,
   size_t length,
   const char * format [, argument_list]
);
int __cdecl _snscanf_s_l(
   const char * input,
   size_t length,
   const char * format,
   locale_t locale [, argument_list]
);
int __cdecl _snwscanf_s(
   const wchar_t * input,
   size_t length,
   const wchar_t * format [, argument_list]
);
int __cdecl _snwscanf_s_l(
   const wchar_t * input,
   size_t length,
   const wchar_t * format,
   locale_t locale [, argument_list]
);
```

Parameters

input
Input string to examine.

length
Number of characters to examine in *input*.

format
One or more format specifiers.

locale
The locale to use.

argument_list
Optional arguments to be assigned according to the format string.

Return Value

Both of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion. For more information, see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

If *input* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter

Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

This function is like **sscanf_s** except that it provides the ability to specify a fixed number of characters to examine from the input string. For more information, see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

The buffer size parameter is required with the type field characters **c**, **C**, **s**, **S**, and **[**. For more information, see scanf Type Field Characters.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_sntscanf_s	**_snscanf_s**	**_snscanf_s**	**_snwscanf_s**
_sntscanf_s_l	**_snscanf_s_l**	**_snscanf_s_l**	**_snwscanf_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_snscanf_s, **_snscanf_s_l**	<stdio.h>
_snwscanf_s, **_snwscanf_s_l**	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_snscanf_s.c
// This example scans a string of
// numbers, using both the character
// and wide character secure versions
// of the snscanf function.

#include <stdio.h>

int main( )
{
    char        str1[] = "15 12 14...";
    wchar_t     str2[] = L"15 12 14...";
    char        s1[3];
    wchar_t     s2[3];
    int         i;
    float       fp;

    i = _snscanf_s( str1, 6,  "%s %f", s1, 3, &fp);
    printf_s("_snscanf_s converted %d fields: ", i);
    printf_s("%s and %f\n", s1, fp);

    i = _snwscanf_s( str2, 6,  L"%s %f", s2, 3, &fp);
    wprintf_s(L"_snwscanf_s converted %d fields: ", i);
    wprintf_s(L"%s and %f\n", s2, fp);
}
```

```
_snscanf_s converted 2 fields: 15 and 12.000000
_snwscanf_s converted 2 fields: 15 and 12.000000
```

See also

scanf Width Specification

sopen

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _sopen or security-enhanced _sopen_s instead.

_sopen, _wsopen

11/8/2018 • 5 minutes to read • Edit Online

Opens a file for sharing. More secure versions of these functions are available: see _sopen_s, _wsopen_s.

Syntax

```
int _sopen(
   const char *filename,
   int oflag,
   int shflag [,
   int pmode ]
);
int _wsopen(
   const wchar_t *filename,
   int oflag,
   int shflag [,
   int pmode ]
);
```

Parameters

filename
File name.

oflag
The kind of operations allowed.

shflag
The kind of sharing allowed.

pmode
Permission setting.

Return Value

Each of these functions returns a file descriptor for the opened file.

If *filename* or *oflag* is a **NULL** pointer, or if *oflag* or *shflag* is not within a valid range of values, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to one of the following values.

ERRNO VALUE	CONDITION
EACCES	The given path is a directory, or the file is read-only, but an open-for-writing operation was attempted.
EEXIST	**_O_CREAT** and **_O_EXCL** flags were specified, but *filename* already exists.
EINVAL	Invalid *oflag* or *shflag* argument.
EMFILE	No more file descriptors are available.

ERRNO VALUE	CONDITION
ENOENT	File or path is not found.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **_sopen** function opens the file specified by *filename* and prepares the file for shared reading or writing, as defined by *oflag* and *shflag*. **_wsopen** is a wide-character version of **_sopen**; the *filename* argument to **_wsopen** is a wide-character string. **_wsopen** and **_sopen** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsopen	**_sopen**	**_sopen**	**_wsopen**

The integer expression *oflag* is formed by combining one or more of the following manifest constants, which are defined in <fcntl.h>. When two or more constants form the argument *oflag*, they are combined with the bitwise-OR operator (**|**).

OFLAG CONSTANT	BEHAVIOR
_O_APPEND	Moves the file pointer to the end of the file before every write operation.
_O_BINARY	Opens the file in binary (untranslated) mode. (See fopen for a description of binary mode.)
_O_CREAT	Creates a file and opens it for writing. Has no effect if the file specified by *filename* exists. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_SHORT_LIVED**	Creates a file as temporary and if possible does not flush to disk. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_TEMPORARY**	Creates a file as temporary; the file is deleted when the last file descriptor is closed. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | `_O_EXCL`	Returns an error value if a file specified by *filename* exists. Applies only when used with **_O_CREAT**.
_O_NOINHERIT	Prevents creation of a shared file descriptor.
_O_RANDOM	Specifies that caching is optimized for, but not restricted to, random access from disk.
_O_RDONLY	Opens a file for reading only. Cannot be specified with **_O_RDWR** or **_O_WRONLY**.
_O_RDWR	Opens a file for both reading and writing. Cannot be specified with **_O_RDONLY** or **_O_WRONLY**.

OFLAG CONSTANT	BEHAVIOR
_O_SEQUENTIAL	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
_O_TEXT	Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)
_O_TRUNC	Opens a file and truncates it to zero length; the file must have write permission. Cannot be specified with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a file. **Note:** The **_O_TRUNC** flag destroys the contents of the specified file.
_O_WRONLY	Opens a file for writing only. Cannot be specified with **_O_RDONLY** or **_O_RDWR**.
_O_U16TEXT	Opens a file in Unicode UTF-16 mode.
_O_U8TEXT	Opens a file in Unicode UTF-8 mode.
_O_WTEXT	Opens a file in Unicode mode.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value for the access mode.

When a file is opened in Unicode mode by using **_O_WTEXT**, **_O_U8TEXT**, or **_O_U16TEXT**, input functions translate the data that's read from the file into UTF-16 data stored as type **wchar_t**. Functions that write to a file opened in Unicode mode expect buffers that contain UTF-16 data stored as type **wchar_t**. If the file is encoded as UTF-8, then UTF-16 data is translated into UTF-8 when it is written, and the file's UTF-8-encoded content is translated into UTF-16 when it is read. An attempt to read or write an odd number of bytes in Unicode mode causes a parameter validation error. To read or write data that's stored in your program as UTF-8, use a text or binary file mode instead of a Unicode mode. You are responsible for any required encoding translation.

If **_sopen** is called with **_O_WRONLY** | **_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it first tries to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it opens the file for writing only and uses the default value for the Unicode mode setting.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, which are defined in <share.h>.

SHFLAG CONSTANT	BEHAVIOR
_SH_DENYRW	Denies read and write access to a file.
_SH_DENYWR	Denies write access to a file.
_SH_DENYRD	Denies read access to a file.
_SH_DENYNO	Permits read and write access.

The *pmode* argument is required only when **_O_CREAT** is specified. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed the first time. Otherwise, *pmode* is

ignored. *pmode* is an integer expression that contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, which are defined in <sys\stat.h>. When both constants are given, they are combined with the bitwise-OR operator. The meaning of *pmode* is as follows.

PMODE	MEANING
_S_IREAD	Only reading permitted.
_S_IWRITE	Writing permitted. (In effect, permits reading and writing.)
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

If write permission is not given, the file is read-only. In the Windows operating system, all files are readable; it is not possible to give write-only permission. Therefore, the modes **_S_IWRITE** and **_S_IREAD** | **_S_IWRITE** are equivalent.

_sopen applies the current file-permission mask to *pmode* before the permissions are set. (See _umask.)

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_sopen	<io.h>	<fcntl.h>, <sys\types.h>, <sys\stat.h>, <share.h>
_wsopen	<io.h> or <wchar.h>	<fcntl.h>, <sys\types.h>, <sys\stat.h>, <share.h>

For more compatibility information, see Compatibility.

Example

See the example for _locking.

See also

Low-Level I/O
_close
_creat, _wcreat
fopen, _wfopen
_fsopen, _wfsopen
_open, _wopen

_sopen_s, _wsopen_s

11/8/2018 • 5 minutes to read • Edit Online

Opens a file for sharing. These versions of _sopen and _wsopen have security enhancements, as described in Security Features in the CRT.

Syntax

```
errno_t _sopen_s(
   int* pfh,
   const char *filename,
   int oflag,
   int shflag,
   int pmode
);
errno_t _wsopen_s(
   int* pfh,
   const wchar_t *filename,
   int oflag,
   int shflag,
   int pmode,
);
```

Parameters

pfh
The file handle, or -1 in the case of an error.

filename
File name.

oflag
The kind of operations allowed.

shflag
The kind of sharing allowed.

pmode
Permission setting.

Return Value

A nonzero return value indicates an error; in that case **errno** is set to one of the following values.

ERRNO VALUE	CONDITION
EACCES	The given path is a directory, or the file is read-only, but an open-for-writing operation was attempted.
EEXIST	**_O_CREAT** and **_O_EXCL** flags were specified, but *filename* already exists.
EINVAL	Invalid *oflag*, *shflag*, or *pmode* argument, or *pfh* or *filename* was a null pointer.

ERRNO VALUE	CONDITION
EMFILE	No more file descriptors available.
ENOENT	File or path not found.

If an invalid argument is passed to the function, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and **EINVAL** is returned.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

In the case of an error, -1 is returned through *pfh* (unless *pfh* is a null pointer).

Remarks

The **_sopen_s** function opens the file specified by *filename* and prepares the file for shared reading or writing, as defined by *oflag* and *shflag*. **_wsopen_s** is a wide-character version of **_sopen_s**; the *filename* argument to **_wsopen_s** is a wide-character string. **_wsopen_s** and **_sopen_s** behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsopen_s	**_sopen_s**	**_sopen_s**	**_wsopen_s**

The integer expression *oflag* is formed by combining one or more manifest constants, which are defined in <fcntl.h>. When two or more constants form the argument *oflag*, they are combined with the bitwise-OR operator (**|**).

OFLAG CONSTANT	BEHAVIOR
_O_APPEND	Moves the file pointer to the end of the file before every write operation.
_O_BINARY	Opens the file in binary (untranslated) mode. (See fopen for a description of binary mode.)
_O_CREAT	Creates a file and opens it for writing. Has no effect if the file specified by *filename* exists. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_SHORT_LIVED**	Creates a file as temporary and if possible does not flush to disk. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | **_O_TEMPORARY**	Creates a file as temporary; the file is deleted when the last file descriptor is closed. The *pmode* argument is required when **_O_CREAT** is specified.
_O_CREAT | `_O_EXCL`	Returns an error value if a file specified by *filename* exists. Applies only when used with **_O_CREAT**.
_O_NOINHERIT	Prevents creation of a shared file descriptor.

OFLAG CONSTANT	BEHAVIOR
_O_RANDOM	Specifies that caching is optimized for, but not restricted to, random access from disk.
_O_RDONLY	Opens a file for reading only. Cannot be specified with **_O_RDWR** or **_O_WRONLY**.
_O_RDWR	Opens a file for both reading and writing. Cannot be specified with **_O_RDONLY** or **_O_WRONLY**.
_O_SEQUENTIAL	Specifies that caching is optimized for, but not restricted to, sequential access from disk.
_O_TEXT	Opens a file in text (translated) mode. (For more information, see Text and Binary Mode File I/O and fopen.)
_O_TRUNC	Opens a file and truncates it to zero length; the file must have write permission. Cannot be specified with **_O_RDONLY**. **_O_TRUNC** used with **_O_CREAT** opens an existing file or creates a file. **Note:** The **_O_TRUNC** flag destroys the contents of the specified file.
_O_WRONLY	Opens a file for writing only. Cannot be specified with **_O_RDONLY** or **_O_RDWR**.
_O_U16TEXT	Opens a file in Unicode UTF-16 mode.
_O_U8TEXT	Opens a file in Unicode UTF-8 mode.
_O_WTEXT	Opens a file in Unicode mode.

To specify the file access mode, you must specify either **_O_RDONLY**, **_O_RDWR**, or **_O_WRONLY**. There is no default value for the access mode.

When a file is opened in Unicode mode by using **_O_WTEXT**, **_O_U8TEXT**, or **_O_U16TEXT**, input functions translate the data that's read from the file into UTF-16 data stored as type **wchar_t**. Functions that write to a file opened in Unicode mode expect buffers that contain UTF-16 data stored as type **wchar_t**. If the file is encoded as UTF-8, then UTF-16 data is translated into UTF-8 when it is written, and the file's UTF-8-encoded content is translated into UTF-16 when it is read. An attempt to read or write an odd number of bytes in Unicode mode causes a parameter validation error. To read or write data that's stored in your program as UTF-8, use a text or binary file mode instead of a Unicode mode. You are responsible for any required encoding translation.

If **_sopen_s** is called with **_O_WRONLY** | **_O_APPEND** (append mode) and **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, it first tries to open the file for reading and writing, read the BOM, then reopen it for writing only. If opening the file for reading and writing fails, it opens the file for writing only and uses the default value for the Unicode mode setting.

The argument *shflag* is a constant expression that consists of one of the following manifest constants, which are defined in <share.h>.

SHFLAG CONSTANT	BEHAVIOR
_SH_DENYRW	Denies read and write access to a file.

SHFLAG CONSTANT	BEHAVIOR
_SH_DENYWR	Denies write access to a file.
_SH_DENYRD	Denies read access to a file.
_SH_DENYNO	Permits read and write access.

The *pmode* argument is always required, unlike in **_sopen**. When you specify **_O_CREAT**, if the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed the first time. Otherwise, *pmode* is ignored. *pmode* is an integer expression that contains one or both of the manifest constants **_S_IWRITE** and **_S_IREAD**, which are defined in <sys\stat.h>. When both constants are given, they are combined with the bitwise-OR operator. The meaning of *pmode* is as follows.

PMODE	MEANING
_S_IREAD	Only reading permitted.
_S_IWRITE	Writing permitted. (In effect, permits reading and writing.)
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

If write permission is not given, the file is read-only. In the Windows operating system, all files are readable; it is not possible to give write-only permission. Therefore, the modes **_S_IWRITE** and **_S_IREAD** | **_S_IWRITE** are equivalent.

_sopen_s applies the current file-permission mask to *pmode* before the permissions are set. (See _umask.)

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_sopen_s	<io.h>	<fcntl.h>, <sys\types.h>, <sys\stat.h>, <share.h>
_wsopen_s	<io.h> or <wchar.h>	<fcntl.h>, <sys/types.h>, <sys/stat.h>, <share.h>

_sopen_s and **_wsopen_s** are Microsoft extensions. For more compatibility information, see Compatibility.

Example

See the example for _locking.

See also

Low-Level I/O
_close
_creat, _wcreat
fopen, _wfopen
_fsopen, _wfsopen
_open, _wopen

spawnl

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnl instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnl, _wspawnl

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnl(
   int mode,
   const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL
);
intptr_t _wspawnl(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

mode
Execution mode for the calling process.

cmdname
Path of the file to be executed.

arg0, *arg1*, ... *argn*
List of pointers to arguments. The *arg0* argument is usually a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

Return Value

The return value from a synchronous **_spawnl** or **_wspawnl** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnl** or **_wspawnl** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter.

Requirements

ROUTINE	REQUIRED HEADER
_spawnl	<process.h>
_wspawnl	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnle

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnle instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnle, _wspawnle

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnle(
   int mode,
   const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wspawnle(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

mode
Execution mode for the calling process.

cmdname
Path of the file to be executed.

arg0, *arg1*, ... *argn*
List of pointers to arguments. The *arg0* argument is usually a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

envp
Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnle** or **_wspawnle** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnle** or **_wspawnle** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an

abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter and also passing an array of pointers to environment settings.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnle	<process.h>
_wspawnle	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnlp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnlp instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnlp, _wspawnlp

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnlp(
   int mode,
   const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL
);
intptr_t _wspawnlp(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL
);
```

Parameters

mode
Execution mode for the calling process.

cmdname
Path of the file to be executed.

arg0, *arg1*, ... *argn*
List of pointers to arguments. The *arg0* argument is usually a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

Return Value

The return value from a synchronous **_spawnlp** or **_wspawnlp** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnlp** or **_wspawnlp** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing each command-line argument as a separate parameter and using the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, these functions generate an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnlp	<process.h>
_wspawnlp	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnlpe

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnlpe instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnlpe, _wspawnlpe

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnlpe(
   int mode,
   const char *cmdname,
   const char *arg0,
   const char *arg1,
   ... const char *argn,
   NULL,
   const char *const *envp
);
intptr_t _wspawnlpe(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *arg0,
   const wchar_t *arg1,
   ... const wchar_t *argn,
   NULL,
   const wchar_t *const *envp
);
```

Parameters

mode
Execution mode for the calling process.

cmdname
Path of the file to be executed.

arg0, *arg1*, ... *argn*
List of pointers to arguments. The *arg0* argument is typically a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings that form the new argument list. Following *argn*, there must be a **NULL** pointer to mark the end of the argument list.

envp
Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnlpe** or **_wspawnlpe** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnlpe** or **_wspawnlpe** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically uses a nonzero argument to call the **exit** routine. If the new process did not explicitly set a positive exit status, a positive exit status indicates an

abnormal exit caused by an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passes each command-line argument as a separate parameter, and passes an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *arg0* is an empty string or a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnlpe	<process.h>
_wspawnlpe	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp

system, _wsystem

spawnv

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnv instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnv, _wspawnv

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnv(
   int mode,
   const char *cmdname,
   const char *const *argv
);
intptr_t _wspawnv(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

mode
Execution mode for the calling process.

cmdname
Path of the file to be executed.

argv
Array of pointers to arguments. The argument *argv*[0] is usually a pointer to a path in real mode or to the program name in protected mode, and *argv*[1] through *argv*[**n**] are pointers to the character strings forming the new argument list. The argument *argv*[**n** +1] must be a **NULL** pointer to mark the end of the argument list.

Return Value

The return value from a synchronous **_spawnv** or **_wspawnv** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnv** or **_wspawnv** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.

ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv*[0] is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnv	<stdio.h> or <process.h>
_wspawnv	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnve

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnve instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnve, _wspawnve

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnve(
   int mode,
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wspawnve(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

mode
Execution mode for a calling process.

cmdname
Path of the file to be executed.

argv
Array of pointers to arguments. The argument *argv*[0] is usually a pointer to a path in real mode or to the program name in protected mode, and *argv*[1] through *argv*[**n**] are pointers to the character strings forming the new argument list. The argument *argv*[**n** +1] must be a **NULL** pointer to mark the end of the argument list.

envp
Array of pointers to environment settings.

Return Value

The return value from a synchronous **_spawnve** or **_wspawnve** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnve** or **_wspawnve** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values.

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these and other return codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv*[0] is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnve	<stdio.h> or <process.h>
_wspawnve	<stdio.h> or <wchar.h>

For more compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnvp

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnvp instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnvp, _wspawnvp

11/8/2018 • 2 minutes to read • Edit Online

Creates a process and executes it.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnvp(
   int mode,
   const char *cmdname,
   const char *const *argv
);
intptr_t _wspawnvp(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv
);
```

Parameters

mode
Execution mode for calling the process.

cmdname
Path of the file to be executed.

argv
Array of pointers to arguments. The argument *argv*[0] is usually a pointer to a path in real mode or to the program name in protected mode, and *argv*[1] through *argv*[**n**] are pointers to the character strings that form the new argument list. The argument *argv*[**n** +1] must be a **NULL** pointer to mark the end of the argument list.

Return Value

The return value from a synchronous **_spawnvp** or **_wspawnvp** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnvp** or **_wspawnvp** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically uses a nonzero argument to call the **exit** routine. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values:

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.

ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

For more information about these, and other, return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions creates a new process and executes it, and passes an array of pointers to command-line arguments and uses the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv*[0] is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnvp	<stdio.h> or <process.h>
_wspawnvp	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

Process and Environment Control
_spawn, _wspawn Functions
abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

spawnvpe

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _spawnvpe instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_spawnvpe, _wspawnvpe

11/8/2018 • 2 minutes to read • Edit Online

Creates and executes a new process.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
intptr_t _spawnvpe(
   int mode,
   const char *cmdname,
   const char *const *argv,
   const char *const *envp
);
intptr_t _wspawnvpe(
   int mode,
   const wchar_t *cmdname,
   const wchar_t *const *argv,
   const wchar_t *const *envp
);
```

Parameters

mode
Execution mode for calling process

cmdname
Path of file to be executed

argv
Array of pointers to arguments. The argument *argv*[0] is usually a pointer to a path in real mode or to the program name in protected mode, and *argv*[1] through *argv*[**n**] are pointers to the character strings forming the new argument list. The argument *argv*[**n** +1] must be a **NULL** pointer to mark the end of the argument list.

envp
Array of pointers to environment settings

Return Value

The return value from a synchronous **_spawnvpe** or **_wspawnvpe** (**_P_WAIT** specified for *mode*) is the exit status of the new process. The return value from an asynchronous **_spawnvpe** or **_wspawnvpe** (**_P_NOWAIT** or **_P_NOWAITO** specified for *mode*) is the process handle. The exit status is 0 if the process terminated normally. You can set the exit status to a nonzero value if the spawned process specifically calls the **exit** routine with a nonzero argument. If the new process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the new process is not started). In this case, **errno** is set to one of the following values:

E2BIG	Argument list exceeds 1024 bytes.
EINVAL	*mode* argument is invalid.
ENOENT	File or path is not found.
ENOEXEC	Specified file is not executable or has invalid executable-file format.
ENOMEM	Not enough memory is available to execute the new process.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

Each of these functions creates and executes a new process, passing an array of pointers to command-line arguments and an array of pointers to environment settings. These functions use the **PATH** environment variable to find the file to execute.

These functions validate their parameters. If either *cmdname* or *argv* is a null pointer, or if *argv* points to null pointer, or *argv*[0] is an empty string, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL**, and return -1. No new process is spawned.

Requirements

ROUTINE	REQUIRED HEADER
_spawnvpe	<stdio.h> or <process.h>
_wspawnvpe	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example in _spawn, _wspawn Functions.

See also

abort
atexit
_exec, _wexec Functions
exit, _Exit, _exit
_flushall
_getmbcp
_onexit, _onexit_m
_setmbcp
system, _wsystem

_splitpath, _wsplitpath

10/31/2018 • 2 minutes to read • Edit Online

Break a path name into components. More secure versions of these functions are available, see _splitpath_s, _wsplitpath_s.

Syntax

```
void _splitpath(
   const char *path,
   char *drive,
   char *dir,
   char *fname,
   char *ext
);
void _wsplitpath(
   const wchar_t *path,
   wchar_t *drive,
   wchar_t *dir,
   wchar_t *fname,
   wchar_t *ext
);
```

Parameters

path
Full path.

drive
Drive letter, followed by a colon (**:**). You can pass **NULL** for this parameter if you do not need the drive letter.

dir
Directory path, including trailing slash. Forward slashes (**/**), backslashes (****), or both may be used. You can pass **NULL** for this parameter if you do not need the directory path.

fname
Base filename (no extension). You can pass **NULL** for this parameter if you do not need the filename.

ext
Filename extension, including leading period (**.**). You can pass **NULL** for this parameter if you do not need the filename extension.

Remarks

The **_splitpath** function breaks a path into its four components. **_splitpath** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wsplitpath** is a wide-character version of **_splitpath**; the arguments to **_wsplitpath** are wide-character strings. These functions behave identically otherwise.

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns. More secure versions of these functions are available; see _splitpath_s, _wsplitpath_s.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsplitpath	**_splitpath**	**_splitpath**	**_wsplitpath**

Each component of the full path is stored in a separate buffer; the manifest constants **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_FNAME**, and **_MAX_EXT** (defined in STDLIB.H) specify the maximum size for each file component. File components that are larger than the corresponding manifest constants cause heap corruption.

Each buffer must be as large as its corresponding manifest constant to avoid potential buffer overrun.

The following table lists the values of the manifest constants.

NAME	VALUE
_MAX_DRIVE	3
_MAX_DIR	256
_MAX_FNAME	256
_MAX_EXT	256

If the full path does not contain a component (for example, a filename), **_splitpath** assigns empty strings to the corresponding buffers.

You can pass **NULL** to **_splitpath** for any parameter other than *path* that you do not need.

If *path* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns **EINVAL**.

Requirements

ROUTINE	REQUIRED HEADER
_splitpath	<stdlib.h>
_wsplitpath	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for _makepath.

See also

File Handling
_fullpath, _wfullpath
_getmbcp
_makepath, _wmakepath
_setmbcp
_splitpath_s, _wsplitpath_s

_splitpath_s, _wsplitpath_s

3/1/2019 • 3 minutes to read • Edit Online

Breaks a path name into components. These are versions of _splitpath, _wsplitpath with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _splitpath_s(
   const char * path,
   char * drive,
   size_t driveNumberOfElements,
   char * dir,
   size_t dirNumberOfElements,
   char * fname,
   size_t nameNumberOfElements,
   char * ext,
   size_t extNumberOfElements
);
errno_t _wsplitpath_s(
   const wchar_t * path,
   wchar_t * drive,
   size_t driveNumberOfElements,
   wchar_t *dir,
   size_t dirNumberOfElements,
   wchar_t * fname,
   size_t nameNumberOfElements,
   wchar_t * ext,
   size_t extNumberOfElements
);
template <size_t drivesize, size_t dirsize, size_t fnamesize, size_t extsize>
errno_t _splitpath_s(
   const char *path,
   char (&drive)[drivesize],
   char (&dir)[dirsize],
   char (&fname)[fnamesize],
   char (&ext)[extsize]
); // C++ only
template <size_t drivesize, size_t dirsize, size_t fnamesize, size_t extsize>
errno_t _wsplitpath_s(
   const wchar_t *path,
   wchar_t (&drive)[drivesize],
   wchar_t (&dir)[dirsize],
   wchar_t (&fname)[fnamesize],
   wchar_t (&ext)[extsize]
); // C++ only
```

Parameters

path
Full path.

drive
Drive letter, followed by a colon (:). You can pass **NULL** for this parameter if you do not need the drive letter.

driveNumberOfElements
The size of the *drive* buffer in single-byte or wide characters. If *drive* is **NULL**, this value must be 0.

dir

Directory path, including trailing slash. Forward slashes (**/**), backslashes (****), or both may be used. You can pass **NULL** for this parameter if you do not need the directory path.

dirNumberOfElements
The size of the *dir* buffer in single-byte or wide characters. If *dir* is **NULL**, this value must be 0.

fname
Base filename (without extension). You can pass **NULL** for this parameter if you do not need the filename.

nameNumberOfElements
The size of the *fname* buffer in single-byte or wide characters. If *fname* is **NULL**, this value must be 0.

ext
Filename extension, including leading period (**.**).You can pass **NULL** for this parameter if you do not need the filename extension.

extNumberOfElements
The size of *ext* buffer in single-byte or wide characters. If *ext* is **NULL**, this value must be 0.

Return Value

Zero if successful; an error code on failure.

Error Conditions

CONDITION	RETURN VALUE
path is **NULL**	**EINVAL**
drive is **NULL**, *driveNumberOfElements* is non-zero	**EINVAL**
drive is non-**NULL**, *driveNumberOfElements* is zero	**EINVAL**
dir is **NULL**, *dirNumberOfElements* is non-zero	**EINVAL**
dir is non-**NULL**, *dirNumberOfElements* is zero	**EINVAL**
fname is **NULL**, *nameNumberOfElements* is non-zero	**EINVAL**
fname is non-**NULL**, *nameNumberOfElements* is zero	**EINVAL**
ext is **NULL**, *extNumberOfElements* is non-zero	**EINVAL**
ext is non-**NULL**, *extNumberOfElements* is zero	**EINVAL**

If any of the above conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

If any of the buffers is too short to hold the result, these functions clear all the buffers to empty strings, set **errno** to **ERANGE**, and return **ERANGE**.

Remarks

The **_splitpath_s** function breaks a path into its four components. **_splitpath_s** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use. **_wsplitpath_s** is a wide-character version of **_splitpath_s**; the arguments

to **_wsplitpath_s** are wide-character strings. These functions behave identically otherwise

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsplitpath_s	**_splitpath_s**	**_splitpath_s**	**_wsplitpath_s**

Each component of the full path is stored in a separate buffer; the manifest constants **_MAX_DRIVE**, **_MAX_DIR**, **_MAX_FNAME**, and **_MAX_EXT** (defined in STDLIB.H) specify the maximum allowable size for each file component. File components larger than the corresponding manifest constants cause heap corruption.

The following table lists the values of the manifest constants.

NAME	VALUE
_MAX_DRIVE	3
_MAX_DIR	256
_MAX_FNAME	256
_MAX_EXT	256

If the full path does not contain a component (for example, a filename), **_splitpath_s** assigns an empty string to the corresponding buffer.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Requirements

ROUTINE	REQUIRED HEADER
_splitpath_s	<stdlib.h>
_wsplitpath_s	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for _makepath_s, _wmakepath_s.

See also

File Handling
_splitpath, _wsplitpath
_fullpath, _wfullpath
_getmbcp

_makepath, _wmakepath
_setmbcp

sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

3/1/2019 • 3 minutes to read • Edit Online

Write formatted data to a string. More secure versions of some of these functions are available; see sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l. The secure versions of **swprintf** and **_swprintf_l** do not take a *count* parameter.

Syntax

```
int sprintf(
   char *buffer,
   const char *format [,
   argument] ...
);
int _sprintf_l(
   char *buffer,
   const char *format,
   locale_t locale [,
   argument] ...
);
int swprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format [,
   argument]...
);
int _swprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
int __swprintf_l(
   wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
   argument] ...
);
template <size_t size>
int sprintf(
   char (&buffer)[size],
   const char *format [,
   argument] ...
); // C++ only
template <size_t size>
int _sprintf_l(
   char (&buffer)[size],
   const char *format,
   locale_t locale [,
   argument] ...
); // C++ only
```

Parameters

buffer

Storage location for output

count

Maximum number of characters to store in the Unicode version of this function.

format
Format-control string

argument
Optional arguments

locale
The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or -1 if an error occurred. If *buffer* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

sprintf returns the number of bytes stored in *buffer*, not counting the terminating null character. **swprintf** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

Remarks

The **sprintf** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. A null character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined.

> **IMPORTANT**
> Using **sprintf**, there is no way to limit the number of characters written, which means that code using **sprintf** is susceptible to buffer overruns. Consider using the related function _snprintf, which specifies a maximum number of characters to be written to *buffer*, or use _scprintf to determine how large a buffer is required. Also, ensure that *format* is not a user-defined string.

swprintf is a wide-character version of **sprintf**; the pointer arguments to **swprintf** are wide-character strings. Detection of encoding errors in **swprintf** may differ from that in **sprintf**. **swprintf** and **fwprintf** behave identically except that **swprintf** writes output to a string rather than to a destination of type **FILE**, and **swprintf** requires the *count* parameter to specify the maximum number of characters to be written. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

swprintf conforms to the ISO C Standard, which requires the second parameter, *count*, of type **size_t**. To force the old nonstandard behavior, define **_CRT_NON_CONFORMING_SWPRINTFS**. In a future version, the old behavior may be removed, so code should be changed to use the new conformant behavior.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_stprintf	**sprintf**	**sprintf**	**_swprintf**
_stprintf_l	**_sprintf_l**	**_sprintf_l**	**__swprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
sprintf, **_sprintf_l**	<stdio.h>
swprintf, **_swprintf_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sprintf.c
// compile with: /W3
// This program uses sprintf to format various
// data and place them in the string named buffer.

#include <stdio.h>

int main( void )
{
   char  buffer[200], s[] = "computer", c = 'l';
   int   i = 35, j;
   float fp = 1.7320534f;

   // Format and print various data:
   j  = sprintf( buffer,     "   String:    %s\n", s ); // C4996
   j += sprintf( buffer + j, "   Character: %c\n", c ); // C4996
   j += sprintf( buffer + j, "   Integer:   %d\n", i ); // C4996
   j += sprintf( buffer + j, "   Real:      %f\n", fp );// C4996
   // Note: sprintf is deprecated; consider using sprintf_s instead

   printf( "Output:\n%s\ncharacter count = %d\n", buffer, j );
}
```

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053

character count = 79
```

Example

```
// crt_swprintf.c
// wide character example
// also demonstrates swprintf returning error code
#include <stdio.h>

int main( void )
{
   wchar_t buf[100];
   int len = swprintf( buf, 100, L"%s", L"Hello world" );
   printf( "wrote %d characters\n", len );
   len = swprintf( buf, 100, L"%s", L"Hello\xffff world" );
   // swprintf fails because string contains WEOF (\xffff)
   printf( "wrote %d characters\n", len );
}
```

```
wrote 11 characters
wrote -1 characters
```

See also

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

10/31/2018 • 3 minutes to read • Edit Online

Write formatted data to a string with the ability to specify the order that the parameters are used in the format string.

Syntax

```
int _sprintf_p(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format [,
   argument_list]
);
int _sprintf_p_l(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format,
   locale_t locale [,
   argument_list]
);
int _swprintf_p(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format [,
   argument_list]
);
int _swprintf_p_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format,
   locale_t locale [,
   argument_list]
);
```

Parameters

buffer
Storage location for output

sizeOfBuffer
Maximum number of characters to store.

format
Format-control string.

argument_list
Optional arguments to the format string.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or -1 if an error occurred.

Remarks

The **_sprintf_p** function formats and stores a series of characters and values in *buffer*. Each argument in the *argument_list* (if any) is converted and output according to the corresponding format specification in *format*. The *format* argument uses the format specification syntax for printf and wprintf functions. A null character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined. The difference between **_sprintf_p** and **sprintf_s** is that **_sprintf_p** supports positional parameters, which allows specifying the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

_swprintf_p is a wide-character version of **_sprintf_p**; the pointer arguments to **_swprintf_p** are wide-character strings. Detection of encoding errors in **_swprintf_p** may differ from that in **_sprintf_p**. **_swprintf_p** and **fwprintf_p** behave identically except that **_swprintf_p** writes output to a string rather than to a destination of type **FILE**, and **_swprintf_p** requires the *count* parameter to specify the maximum number of characters to be written. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

_sprintf_p returns the number of bytes stored in *buffer*, not counting the terminating null character. **_swprintf_p** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character. If *buffer* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_stprintf_p	**_sprintf_p**	**_sprintf_p**	**_swprintf_p**
_stprintf_p_l	**_sprintf_p_l**	**_sprintf_p_l**	**_swprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_sprintf_p, **_sprintf_p_l**	<stdio.h>
_swprintf_p, **_swprintf_p_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sprintf_p.c
// This program uses _sprintf_p to format various
// data and place them in the string named buffer.
//

#include <stdio.h>

int main( void )
{
    char     buffer[200],
            s[] = "computer", c = 'l';
    int      i = 35,
            j;
    float    fp = 1.7320534f;

    // Format and print various data:
    j  = _sprintf_p( buffer, 200,
                     "   String:    %s\n", s );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Character: %c\n", c );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Integer:   %d\n", i );
    j += _sprintf_p( buffer + j, 200 - j,
                     "   Real:      %f\n", fp );

    printf( "Output:\n%s\ncharacter count = %d\n",
            buffer, j );
}
```

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053

character count = 79
```

Example

```
// crt_swprintf_p.c
// This is the wide character example which
// also demonstrates _swprintf_p returning
// error code.
#include <stdio.h>

#define BUFFER_SIZE 100

int main( void )
{
    wchar_t buffer[BUFFER_SIZE];
    int     len;

    len = _swprintf_p(buffer, BUFFER_SIZE, L"%2$s %1$d",
                      0, L" marbles in your head.");
    _printf_p( "Wrote %d characters\n", len );

    // _swprintf_p fails because string contains WEOF (\xffff)
    len = _swprintf_p(buffer, BUFFER_SIZE, L"%s",
                      L"Hello\xffff world" );
    _printf_p( "Wrote %d characters\n", len );
}
```

```
Wrote 24 characters
Wrote -1 characters
```

See also

Stream I/O
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l
vprintf Functions
printf_p Positional Parameters

sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l

3/1/2019 • 3 minutes to read • Edit Online

Write formatted data to a string. These are versions of sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int sprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format,
   ...
);
int _sprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   const char *format,
   locale_t locale,
   ...
);
int swprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format,
   ...
);
int _swprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   const wchar_t *format,
   locale_t locale,
   ...
);
template <size_t size>
int sprintf_s(
   char (&buffer)[size],
   const char *format,
   ...
); // C++ only
template <size_t size>
int swprintf_s(
   wchar_t (&buffer)[size],
   const wchar_t *format,
   ...
); // C++ only
```

Parameters

buffer
Storage location for output

sizeOfBuffer
Maximum number of characters to store.

format
Format-control string

...

Optional arguments to be formatted

locale
The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or -1 if an error occurred. If *buffer* or *format* is a null pointer, **sprintf_s** and **swprintf_s** return -1 and set **errno** to **EINVAL**.

sprintf_s returns the number of bytes stored in *buffer*, not counting the terminating null character. **swprintf_s** returns the number of wide characters stored in *buffer*, not counting the terminating null wide character.

Remarks

The **sprintf_s** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf. A null character is appended after the last character written. If copying occurs between strings that overlap, the behavior is undefined.

One main difference between **sprintf_s** and sprintf is that **sprintf_s** checks the format string for valid formatting characters, whereas sprintf only checks if the format string or buffer are **NULL** pointers. If either check fails, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and sets **errno** to **EINVAL**.

The other main difference between **sprintf_s** and sprintf is that **sprintf_s** takes a length parameter specifying the size of the output buffer in characters. If the buffer is too small for the formatted text, including the terminating null, then the buffer is set to an empty string by placing a null character at *buffer*[0], and the invalid parameter handler is invoked. Unlike **_snprintf**, **sprintf_s** guarantees that the buffer will be null-terminated unless the buffer size is zero.

swprintf_s is a wide-character version of **sprintf_s**; the pointer arguments to **swprintf_s** are wide-character strings. Detection of encoding errors in **swprintf_s** may differ from that in **sprintf_s**. The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, use of these functions is simplified by template overloads; the overloads can infer buffer length automatically, which eliminates the need to specify a size argument, and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

There are versions of **sprintf_s** that offer additional control over what happens if the buffer is too small. For more information, see _snprintf_s, _snprintf_s_l, _snwprintf_s, _snwprintf_s_l.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_stprintf_s	**sprintf_s**	**sprintf_s**	**swprintf_s**
_stprintf_s_l	**_sprintf_s_l**	**_sprintf_s_l**	**_swprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER
sprintf_s, **_sprintf_s_l**	C: <stdio.h> C++: <cstdio> or <stdio.h>
swprintf_s, **_swprintf_s_l**	C: <stdio.h> or <wchar.h> C++: <cstdio>, <cwchar>, <stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sprintf_s.c
// This program uses sprintf_s to format various
// data and place them in the string named buffer.
//

#include <stdio.h>

int main( void )
{
   char  buffer[200], s[] = "computer", c = 'l';
   int   i = 35, j;
   float fp = 1.7320534f;

   // Format and print various data:
   j  = sprintf_s( buffer, 200,     "   String:    %s\n", s );
   j += sprintf_s( buffer + j, 200 - j, "   Character: %c\n", c );
   j += sprintf_s( buffer + j, 200 - j, "   Integer:   %d\n", i );
   j += sprintf_s( buffer + j, 200 - j, "   Real:      %f\n", fp );

   printf_s( "Output:\n%s\ncharacter count = %d\n", buffer, j );
}
```

```
Output:
   String:    computer
   Character: l
   Integer:   35
   Real:      1.732053

character count = 79
```

Example

```
// crt_swprintf_s.c
// wide character example
// also demonstrates swprintf_s returning error code
#include <stdio.h>

int main( void )
{
   wchar_t buf[100];
   int len = swprintf_s( buf, 100, L"%s", L"Hello world" );
   printf( "wrote %d characters\n", len );
   len = swprintf_s( buf, 100, L"%s", L"Hello\xffff world" );
   // swprintf_s fails because string contains WEOF (\xffff)
   printf( "wrote %d characters\n", len );
}
```

```
wrote 11 characters
wrote -1 characters
```

See also

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

sqrt, sqrtf, sqrtl

3/1/2019 • 2 minutes to read • Edit Online

Calculates the square root.

Syntax

```
double sqrt(
   double x
);
float sqrt(
   float x
);  // C++ only
long double sqrt(
   long double x
);  // C++ only
float sqrtf(
   float x
);
long double sqrtl(
   long double x
);
```

Parameters

x

Non-negative floating-point value

Remarks

Because C++ allows overloading, you can call overloads of **sqrt** that take **float** or **long double** types. In a C program, **sqrt** always takes and returns **double**.

Return Value

The **sqrt** functions return the square-root of *x*. By default, if *x* is negative, **sqrt** returns an indefinite NaN.

INPUT	SEH EXCEPTION	_MATHERR EXCEPTION
± QNAN,IND	none	_DOMAIN
- ∞	none	_DOMAIN
x<0	none	_DOMAIN

Requirements

FUNCTION	C HEADER	C++ HEADER
sqrt, **sqrtf**, **sqrtl**	<math.h>	<cmath>

For compatibility information, see Compatibility.

Example

```
// crt_sqrt.c
// This program calculates a square root.

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   double question = 45.35, answer;
   answer = sqrt( question );
   if( question < 0 )
      printf( "Error: sqrt returns %f\n", answer );
   else
      printf( "The square root of %.2f is %.2f\n", question, answer );
}
```

```
The square root of 45.35 is 6.73
```

See also

Floating-Point Support
exp, expf, expl
log, logf, log10, log10f
pow, powf, powl
_CIsqrt

srand

3/1/2019 • 2 minutes to read • Edit Online

Sets the starting seed value for the pseudorandom number generator used by the **rand** function.

Syntax

```
void srand(
   unsigned int seed
);
```

Parameters

seed
Seed for pseudorandom number generation

Remarks

The **srand** function sets the starting point for generating a series of pseudorandom integers in the current thread. To reinitialize the generator to create the same sequence of results, call the **srand** function and use the same *seed* argument again. Any other value for *seed* sets the generator to a different starting point in the pseudorandom sequence. **rand** retrieves the pseudorandom numbers that are generated. Calling **rand** before any call to **srand** generates the same sequence as calling **srand** with *seed* passed as 1.

Requirements

ROUTINE	REQUIRED HEADER
srand	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

See the example for rand.

See also

Floating-Point Support
rand

sscanf, _sscanf_l, swscanf, _swscanf_l

10/31/2018 • 2 minutes to read • Edit Online

Read formatted data from a string. More secure versions of these functions are available; see sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l.

Syntax

```
int sscanf(
   const char *buffer,
   const char *format [,
   argument ] ...
);
int _sscanf_l(
   const char *buffer,
   const char *format,
   locale_t locale [,
   argument ] ...
);
int swscanf(
   const wchar_t *buffer,
   const wchar_t *format [,
   argument ] ...
);
int _swscanf_l(
   const wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
   argument ] ...
);
```

Parameters

buffer
Stored data

format
Format-control string. For more information, see Format Specifications.

argument
Optional arguments

locale
The locale to use

Return Value

Each of these functions returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **sscanf** function reads data from *buffer* into the location given by each *argument*. Every *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function. If copying takes place between strings that overlap, the behavior is undefined.

> **IMPORTANT**
> When reading a string with **sscanf**, always specify a width for the **%s** format (for example, **"%32s"** instead of **"%s"**); otherwise, improperly formatted input can easily cause a buffer overrun.

swscanf is a wide-character version of **sscanf**; the arguments to **swscanf** are wide-character strings. **sscanf** does not handle multibyte hexadecimal characters. **swscanf** does not handle Unicode full-width hexadecimal or "compatibility zone" characters. Otherwise, **swscanf** and **sscanf** behave identically.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_stscanf	**sscanf**	**sscanf**	**swscanf**
_stscanf_l	**_sscanf_l**	**_sscanf_l**	**_swscanf_l**

Requirements

ROUTINE	REQUIRED HEADER
sscanf, **_sscanf_l**	<stdio.h>
swscanf, **_swscanf_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sscanf.c
// compile with: /W3
// This program uses sscanf to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>

int main( void )
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;

   // Input various data from tokenstring:
   // max 80 character string:
   sscanf( tokenstring, "%80s", s ); // C4996
   sscanf( tokenstring, "%c", &c );  // C4996
   sscanf( tokenstring, "%d", &i );  // C4996
   sscanf( tokenstring, "%f", &fp ); // C4996
   // Note: sscanf is deprecated; consider using sscanf_s instead

   // Output the data read
   printf( "String    = %s\n", s );
   printf( "Character = %c\n", c );
   printf( "Integer:  = %d\n", i );
   printf( "Real:     = %f\n", fp );
}
```

```
String    = 15
Character = 1
Integer:  = 15
Real:     = 15.000000
```

See also

Stream I/O
fscanf, _fscanf_l, fwscanf, _fwscanf_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l

sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

3/1/2019 • 3 minutes to read • Edit Online

Reads formatted data from a string. These versions of sscanf, _sscanf_l, swscanf, _swscanf_l have security enhancements, as described in Security Features in the CRT.

Syntax

```
int sscanf_s(
   const char *buffer,
   const char *format [,
   argument ] ...
);
int _sscanf_s_l(
   const char *buffer,
   const char *format,
   locale_t locale [,
   argument ] ...
);
int swscanf_s(
   const wchar_t *buffer,
   const wchar_t *format [,
   argument ] ...
);
int _swscanf_s_l(
   const wchar_t *buffer,
   const wchar_t *format,
   locale_t locale [,
   argument ] ...
);
```

Parameters

buffer
Stored data

format
Format-control string. For more information, see Format Specification Fields: scanf and wscanf Functions.

argument
Optional arguments

locale
The locale to use

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **sscanf_s** function reads data from *buffer* into the location that's given by each *argument*. The arguments after the format string specify pointers to variables that have a type that corresponds to a type specifier in *format*. Unlike the less secure version sscanf, a buffer size parameter is required when you use the type field characters **c**, **C**, **s**, **S**, or string control sets that are enclosed in **[]**. The buffer size in characters must be supplied as an additional parameter immediately after each buffer parameter that requires it. For example, if you are reading into a string, the buffer size for that string is passed as follows:

```
wchar_t ws[10];
swscanf_s(in_str, L"%9s", ws, (unsigned)_countof(ws)); // buffer size is 10, width specification is 9
```

The buffer size includes the terminating null. A width specification field may be used to ensure that the token that's read in will fit into the buffer. If no width specification field is used, and the token read in is too big to fit in the buffer, nothing is written to that buffer.

In the case of characters, a single character may be read as follows:

```
wchar_t wc;
swscanf_s(in_str, L"%c", &wc, 1);
```

This example reads a single character from the input string and then stores it in a wide-character buffer. When you read multiple characters for non-null terminated strings, unsigned integers are used as the width specification and the buffer size.

```
char c[4];
sscanf_s(input, "%4c", &c, (unsigned)_countof(c)); // not null terminated
```

For more information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Type Field Characters.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**. When compiling for 64-bit targets, use a static cast to convert **_countof** or **sizeof** results to the correct size.

The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf_s** function. If copying occurs between strings that overlap, the behavior is undefined.

swscanf_s is a wide-character version of **sscanf_s**; the arguments to **swscanf_s** are wide-character strings. **sscanf_s** does not handle multibyte hexadecimal characters. **swscanf_s** does not handle Unicode full-width hexadecimal or "compatibility zone" characters. Otherwise, **swscanf_s** and **sscanf_s** behave identically.

The versions of these functions that have the **_l** suffix are identical except that they use the locale parameter that's passed in instead of the current thread locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_stscanf_s	**sscanf_s**	**sscanf_s**	**swscanf_s**
_stscanf_s_l	**_sscanf_s_l**	**_sscanf_s_l**	**_swscanf_s_l**

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED

Requirements

ROUTINE	REQUIRED HEADER
sscanf_s, **_sscanf_s_l**	<stdio.h>
swscanf_s, **_swscanf_s_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_sscanf_s.c
// This program uses sscanf_s to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;

   // Input various data from tokenstring:
   // max 80 character string plus null terminator
   sscanf_s( tokenstring, "%s", s, (unsigned)_countof(s) );
   sscanf_s( tokenstring, "%c", &c, (unsigned)sizeof(char) );
   sscanf_s( tokenstring, "%d", &i );
   sscanf_s( tokenstring, "%f", &fp );

   // Output the data read
   printf_s( "String    = %s\n", s );
   printf_s( "Character = %c\n", c );
   printf_s( "Integer:  = %d\n", i );
   printf_s( "Real:     = %f\n", fp );
}
```

```
String    = 15
Character = 1
Integer:  = 15
Real:     = 15.000000
```

See also

Stream I/O
fscanf, _fscanf_l, fwscanf, _fwscanf_l
scanf, _scanf_l, wscanf, _wscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l

_stat, _stat32, _stat64, _stati64, _stat32i64, _stat64i32, _wstat, _wstat32, _wstat64, _wstati64, _wstat32i64, _wstat64i32

11/8/2018 • 5 minutes to read • Edit Online

Get status information on a file.

Syntax

```
int _stat(
   const char *path,
   struct _stat *buffer
);
int _stat32(
   const char *path,
   struct __stat32 *buffer
);
int _stat64(
   const char *path,
   struct __stat64 *buffer
);
int _stati64(
   const char *path,
   struct _stati64 *buffer
);
int _stat32i64(
   const char *path,
   struct _stat32i64 *buffer
);
int _stat64i32(
   const char *path,
   struct _stat64i32 *buffer
);
int _wstat(
   const wchar_t *path,
   struct _stat *buffer
);
int _wstat32(
   const wchar_t *path,
   struct __stat32 *buffer
);
int _wstat64(
   const wchar_t *path,
   struct __stat64 *buffer
);
int _wstati64(
   const wchar_t *path,
   struct _stati64 *buffer
);
int _wstat32i64(
   const wchar_t *path,
   struct _stat32i64 *buffer
);
int _wstat64i32(
   const wchar_t *path,
   struct _stat64i32 *buffer
);
```

Parameters

path
Pointer to a string containing the path of existing file or directory.

buffer
Pointer to structure that stores results.

Return Value

Each of these functions returns 0 if the file-status information is obtained. A return value of -1 indicates an error, in which case **errno** is set to **ENOENT**, indicating that the filename or path could not be found. A return value of **EINVAL** indicates an invalid parameter; **errno** is also set to **EINVAL** in this case.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

The date stamp on a file can be represented if it is later than midnight, January 1, 1970, and before 23:59:59, December 31, 3000, UTC, unless you use **_stat32** or **_wstat32**, or have defined **_USE_32BIT_TIME_T**, in which case the date can be represented only until 23:59:59 January 18, 2038, UTC.

Remarks

The **_stat** function obtains information about the file or directory specified by *path* and stores it in the structure pointed to by *buffer*. **_stat** automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the multibyte code page currently in use.

_wstat is a wide-character version of **_stat**; the *path* argument to **_wstat** is a wide-character string. **_wstat** and **_stat** behave identically except that **_wstat** does not handle multibyte-character strings.

Variations of these functions support 32- or 64-bit time types, and 32- or 64-bit file lengths. The first numerical suffix (**32** or **64**) indicates the size of the time type used; the second suffix is either **i32** or **i64**, indicating whether the file size is represented as a 32-bit or 64-bit integer.

_stat is equivalent to **_stat64i32**, and **struct _stat** contains a 64-bit time. This is true unless **_USE_32BIT_TIME_T** is defined, in which case the old behavior is in effect; **_stat** uses a 32-bit time, and **struct _stat** contains a 32-bit time. The same is true for **_stati64**.

> **NOTE**
> **_wstat** does not work with Windows Vista symbolic links. In these cases, **_wstat** will always report a file size of 0. **_stat** does work correctly with symbolic links.

This function validates its parameters. If either *path* or *buffer* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation.

Time Type and File Length Type Variations of _stat

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_stat, **_wstat**	Not defined	64-bit	32-bit
_stat, **_wstat**	Defined	32-bit	32-bit
_stat32, **_wstat32**	Not affected by the macro definition	32-bit	32-bit

FUNCTIONS	_USE_32BIT_TIME_T DEFINED?	TIME TYPE	FILE LENGTH TYPE
_stat64, **_wstat64**	Not affected by the macro definition	64-bit	64-bit
_stati64, **_wstati64**	Not defined	64-bit	64-bit
_stati64, **_wstati64**	Defined	32-bit	64-bit
_stat32i64, **_wstat32i64**	Not affected by the macro definition	32-bit	64-bit
_stat64i32, **_wstat64i32**	Not affected by the macro definition	64-bit	32-bit

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstat	**_stat**	**_stat**	**_wstat**
_tstat64	**_stat64**	**_stat64**	**_wstat64**
_tstati64	**_stati64**	**_stati64**	**_wstati64**
_tstat32i64	**_stat32i64**	**_stat32i64**	**_wstat32i64**
_tstat64i32	**_stat64i32**	**_stat64i32**	**_wstat64i32**

The **_stat** structure, defined in SYS\STAT.H, includes the following fields.

FIELD	
st_gid	Numeric identifier of group that owns the file (UNIX-specific) This field will always be zero on Windows systems. A redirected file is classified as a Windows file.
st_atime	Time of last access of file. Valid on NTFS but not on FAT formatted disk drives.
st_ctime	Time of creation of file. Valid on NTFS but not on FAT formatted disk drives.
st_dev	Drive number of the disk containing the file (same as **st_rdev**).
st_ino	Number of the information node (the **inode**) for the file (UNIX-specific). On UNIX file systems, the **inode** describes the file date and time stamps, permissions, and content. When files are hard-linked to one another, they share the same **inode**. The **inode**, and therefore **st_ino**, has no meaning in the FAT, HPFS, or NTFS file systems.

FIELD	
st_mode	Bit mask for file-mode information. The **_S_IFDIR** bit is set if *path* specifies a directory; the **_S_IFREG** bit is set if *path* specifies an ordinary file or a device. User read/write bits are set according to the file's permission mode; user execute bits are set according to the filename extension.
st_mtime	Time of last modification of file.
st_nlink	Always 1 on non-NTFS file systems.
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_size	Size of the file in bytes; a 64-bit integer for variations with the **i64** suffix.
st_uid	Numeric identifier of user who owns file (UNIX-specific). This field will always be zero on Windows systems. A redirected file is classified as a Windows file.

If *path* refers to a device, the **st_size**, various time fields, **st_dev**, and **st_rdev** fields in the **_stat** structure are meaningless. Because STAT.H uses the _dev_t type that is defined in TYPES.H, you must include TYPES.H before STAT.H in your code.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_stat, **_stat32**, **_stat64**, **_stati64**, **_stat32i64**, **_stat64i32**	<sys/types.h> followed by <sys/stat.h>	<errno.h>
_wstat, **_wstat32**, **_wstat64**, **_wstati64**, **_wstat32i64**, **_wstat64i32**	<sys/types.h> followed by <sys/stat.h> or <wchar.h>	<errno.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_stat.c
// This program uses the _stat function to
// report information about the file named crt_stat.c.

#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>

int main( void )
{
   struct _stat buf;
   int result;
   char timebuf[26];
   char* filename = "crt_stat.c";
   errno_t err;

   // Get data associated with "crt_stat.c":
   result = _stat( filename, &buf );

   // Check if statistics are valid:
   if( result != 0 )
   {
      perror( "Problem getting information" );
      switch (errno)
      {
         case ENOENT:
           printf("File %s not found.\n", filename);
           break;
         case EINVAL:
           printf("Invalid parameter to _stat.\n");
           break;
         default:
           /* Should never be reached. */
           printf("Unexpected error in _stat.\n");
      }
   }
   else
   {
      // Output some of the statistics:
      printf( "File size     : %ld\n", buf.st_size );
      printf( "Drive         : %c:\n", buf.st_dev + 'A' );
      err = ctime_s(timebuf, 26, &buf.st_mtime);
      if (err)
      {
         printf("Invalid arguments to ctime_s.");
         exit(1);
      }
      printf( "Time modified : %s", timebuf );
   }
}
```

```
File size     : 732
Drive         : C:
Time modified : Thu Feb 07 14:39:36 2002
```

See also

File Handling

_access, _waccess

_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32

_getmbcp

_setmbcp

_STATIC_ASSERT Macro

10/31/2018 • 2 minutes to read • Edit Online

Evaluate an expression at compile time and generate an error when the result is **FALSE**.

Syntax

```
_STATIC_ASSERT(
    booleanExpression
);
```

Parameters

booleanExpression
Expression (including pointers) that evaluates to nonzero (**TRUE**) or 0 (**FALSE**).

Remarks

This macro resembles the _ASSERT and _ASSERTE macros, except that *booleanExpression* is evaluated at compile time instead of at runtime. If *booleanExpression* evaluates to **FALSE** (0), Compiler Error C2466 is generated.

Example

In this example, we check whether the sizeof an **int** is larger than or equal to 2 bytes and whether the sizeof a **long** is 1 byte. The program will not compile and it will generate Compiler Error C2466 because a **long** is larger than 1 byte.

```
// crt__static_assert.c

#include <crtdbg.h>
#include <stdio.h>

_STATIC_ASSERT(sizeof(int) >= 2);
_STATIC_ASSERT(sizeof(long) == 1);  // C2466

int main()
{
    printf("I am sure that sizeof(int) will be >= 2: %d\n",
        sizeof(int));
    printf("I am not so sure that sizeof(long) == 1: %d\n",
        sizeof(long));
}
```

Requirements

MACRO	REQUIRED HEADER
_STATIC_ASSERT	<crtdbg.h>

See also

_ASSERT, _ASSERTE, _ASSERT_EXPR Macros

_status87, _statusfp, _statusfp2

10/31/2018 • 2 minutes to read • Edit Online

Gets the floating-point status word.

Syntax

```
unsigned int _status87( void );
unsigned int _statusfp( void );
void _statusfp2(unsigned int *px86, unsigned int *pSSE2)
```

Parameters

px86
This address is filled with the status word for the x87 floating-point unit.

pSSE2
This address is filled with the status word for the SSE2 floating-point unit.

Return Value

For **_status87** and **_statusfp**, the bits in the value that's returned indicate the floating-point status. See the FLOAT.H include file for a definition of the bits that are returned by **_statusfp**. Many math library functions modify the floating-point status word, with unpredictable results. Optimization can reorder, combine, and eliminate floating-point operations around calls to **_status87**, **_statusfp**, and related functions. Use the /Od (Disable (Debug)) compiler option or the fenv_access pragma directive to prevent optimizations that reorder floating-point operations. Return values from **_clearfp** and **_statusfp**, and also the return parameters of **_statusfp2**, are more reliable if fewer floating-point operations are performed between known states of the floating-point status word.

Remarks

The **_statusfp** function gets the floating-point status word. The status word is a combination of the floating-point processor status and other conditions detected by the floating-point exception handler—for example, floating-point stack overflow and underflow. Unmasked exceptions are checked for before the contents of the status word are returned. This means that the caller is informed of pending exceptions. On x86 platforms, **_statusfp** returns a combination of the x87 and SSE2 floating-point status. On x64 platforms, the status that's returned is based on the SSE's MXCSR status. On ARM platforms, **_statusfp** returns status from the FPSCR register.

_statusfp is a platform-independent, portable version of **_status87**. It is identical to **_status87** on Intel (x86) platforms and is also supported by the x64 and ARM platforms. To ensure that your floating-point code is portable to all architectures, use **_statusfp**. If you are only targeting x86 platforms, you can use either **_status87** or **_statusfp**.

We recommend **_statusfp2** for chips (such as the Pentium IV) that have both an x87 and an SSE2 floating-point processor. For **_statusfp2**, the addresses are filled by using the floating-point status word for both the x87 or the SSE2 floating-point processor. For a chip that supports x87 and SSE2 floating-point processors, EM_AMBIGUOUS is set to 1 if **_statusfp** or **_controlfp** is used and the action was ambiguous because it could refer to the x87 or the SSE2 floating-point status word. The **_statusfp2** function is only supported on x86 platforms.

These functions are not useful for /clr (Common Language Runtime Compilation) because the common language runtime (CLR) only supports the default floating-point precision.

Requirements

ROUTINE	REQUIRED HEADER
_status87, **_statusfp**, **_statusfp2**	<float.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_statusfp.c
// Build by using: cl /W4 /Ox /nologo crt_statusfp.c
// This program creates various floating-point errors and
// then uses _statusfp to display messages that indicate these problems.

#include <stdio.h>
#include <float.h>
#pragma fenv_access(on)

double test( void )
{
   double a = 1e-40;
   float b;
   double c;

   printf("Status = 0x%.8x - clear\n", _statusfp());

   // Assignment into b is inexact & underflows:
   b = (float)(a + 1e-40);
   printf("Status = 0x%.8x - inexact, underflow\n", _statusfp());

   // c is denormal:
   c = b / 2.0;
   printf("Status = 0x%.8x - inexact, underflow, denormal\n",
            _statusfp());

   // Clear floating point status:
   _clearfp();
   return c;
}

int main(void)
{
   return (int)test();
}
```

```
Status = 0x00000000 - clear
Status = 0x00000003 - inexact, underflow
Status = 0x00080003 - inexact, underflow, denormal
```

See also

Floating-Point Support
_clear87, _clearfp
_control87, _controlfp, __control87_2

strcat, wcscat, _mbscat

3/1/2019 • 2 minutes to read • Edit Online

Appends a string. More secure versions of these functions are available; see strcat_s, wcscat_s, _mbscat_s.

> **IMPORTANT**
>
> **_mbscat_s** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strcat(
   char *strDestination,
   const char *strSource
);
wchar_t *wcscat(
   wchar_t *strDestination,
   const wchar_t *strSource
);
unsigned char *_mbscat(
   unsigned char *strDestination,
   const unsigned char *strSource
);
template <size_t size>
char *strcat(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
wchar_t *wcscat(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
unsigned char *_mbscat(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination
Null-terminated destination string.

strSource
Null-terminated source string.

Return Value

Each of these functions returns the destination string (*strDestination*). No return value is reserved to indicate an error.

Remarks

The **strcat** function appends *strSource* to *strDestination* and terminates the resulting string with a null character. The initial character of *strSource* overwrites the terminating null character of *strDestination*. The behavior of **strcat** is undefined if the source and destination strings overlap.

> **IMPORTANT**
>
> Because **strcat** does not check for sufficient space in *strDestination* before appending *strSource*, it is a potential cause of buffer overruns. Consider using strncat instead.

wcscat and **_mbscat** are wide-character and multibyte-character versions of **strcat**. The arguments and return value of **wcscat** are wide-character strings; those of **_mbscat** are multibyte-character strings. These three functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscat	**strcat**	**_mbscat**	**wcscat**

Requirements

ROUTINE	REQUIRED HEADER
strcat	<string.h>
wcscat	<string.h> or <wchar.h>
_mbscat	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

See the example for strcpy.

See also

String Manipulation
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strcat_s, wcscat_s, _mbscat_s, _mbscat_s_l

3/1/2019 • 2 minutes to read • Edit Online

Appends a string. These versions of strcat, wcscat, _mbscat have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbscat_s** and **_mbscat_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t strcat_s(
   char *strDestination,
   size_t numberOfElements,
   const char *strSource
);
errno_t wcscat_s(
   wchar_t *strDestination,
   size_t numberOfElements,
   const wchar_t *strSource
);
errno_t _mbscat_s(
   unsigned char *strDestination,
   size_t numberOfElements,
   const unsigned char *strSource
);
errno_t _mbscat_s_l(
   unsigned char *strDestination,
   size_t numberOfElements,
   const unsigned char *strSource,
   _locale_t locale
);
template <size_t size>
errno_t strcat_s(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
errno_t wcscat_s(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
errno_t _mbscat_s(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
template <size_t size>
errno_t _mbscat_s_l(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource,
   _locale_t locale
); // C++ only
```

Parameters

strDestination
Null-terminated destination string buffer.

numberOfElements
Size of the destination string buffer.

strSource
Null-terminated source string buffer.

locale
Locale to use.

Return Value

Zero if successful; an error code on failure.

Error Conditions

STRDESTINATION	*NUMBEROFELEMENTS*	*STRSOURCE*	RETURN VALUE	CONTENTS OF *STRDESTINATION*
NULL or unterminated	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	*strDestination*[0] set to 0
any	0, or too small	any	**ERANGE**	*strDestination*[0] set to 0

Remarks

The **strcat_s** function appends *strSource* to *strDestination* and terminates the resulting string with a null character. The initial character of *strSource* overwrites the terminating null character of *strDestination*. The behavior of **strcat_s** is undefined if the source and destination strings overlap.

Note that the second parameter is the total size of the buffer, not the remaining size:

```
char buf[16];
strcpy_s(buf, 16, "Start");
strcat_s(buf, 16, " End");               // Correct
strcat_s(buf, 16 - strlen(buf), " End"); // Incorrect
```

wcscat_s and **_mbscat_s** are wide-character and multibyte-character versions of **strcat_s**. The arguments and return value of **wcscat_s** are wide-character strings; those of **_mbscat_s** are multibyte-character strings. These three functions behave identically otherwise.

If *strDestination* is a null pointer, or is not null-terminated, or if *strSource* is a **NULL** pointer, or if the destination string is too small, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

The versions of functions that have the **_l** suffix have the same behavior, but use the locale parameter that's passed in instead of the current locale. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscat_s	**strcat_s**	**_mbscat_s**	**wcscat_s**

Requirements

ROUTINE	REQUIRED HEADER
strcat_s	<string.h>
wcscat_s	<string.h> or <wchar.h>
_mbscat_s	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

See the code example in strcpy_s, wcscpy_s, _mbscpy_s.

See also

String Manipulation
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strchr, wcschr, _mbschr, _mbschr_l

10/31/2018 • 3 minutes to read • Edit Online

Finds a character in a string, by using the current locale or a specified LC_CTYPE conversion-state category.

IMPORTANT

`_mbschr` and `_mbschr_l` cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strchr(
   const char *str,
   int c
);  // C only
char *strchr(
   char * str,
   int c
); // C++ only
const char *strchr(
   const char * str,
   int c
); // C++ only
wchar_t *wcschr(
   const wchar_t *str,
   wchar_t c
); // C only
wchar_t *wcschr(
   wchar_t *str,
   wchar_t c
);  // C++ only
const wchar_t *wcschr(
   const wchar_t *str,
   wchar_t c
);  // C++ only
unsigned char *_mbschr(
   const unsigned char *str,
   unsigned int c
); // C only
unsigned char *_mbschr(
   unsigned char *str,
   unsigned int c
); // C++ only
const unsigned char *_mbschr(
   const unsigned char *str,
   unsigned int c
); // C++ only
unsigned char *_mbschr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C only
unsigned char *_mbschr_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
const unsigned char *_mbschr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
```

Parameters

str
Null-terminated source string.

c
Character to be located.

locale
Locale to use.

Return Value

Each of these functions returns a pointer to the first occurrence of *c* in *str*, or NULL if *c* is not found.

Remarks

The `strchr` function finds the first occurrence of *c* in *str*, or it returns NULL if *c* is not found. The null terminating character is included in the search.

`wcschr`, `_mbschr` and `_mbschr_l` are wide-character and multibyte-character versions of `strchr`. The arguments and return value of `wcschr` are wide-character strings; those of `_mbschr` are multibyte-character strings. `_mbschr` recognizes multibyte-character sequences. Also, if the string is a null pointer, `_mbschr` invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, `_mbschr` returns NULL and sets `errno` to EINVAL. `strchr` and `wcschr` do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the LC_CTYPE category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CRT_CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tcschr`	`strchr`	`_mbschr`	`wcschr`
_n/a	**n/a**	`_mbschr_l`	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
`strchr`	<string.h>
`wcschr`	<string.h> or <wchar.h>
`_mbschr`, `_mbschr_l`	<mbstring.h>

For more information about compatibility, see Compatibility.

Example

```
// crt_strchr.c
//
// This program illustrates searching for a character
// with strchr (search forward) or strrchr (search backward).
//

#include <string.h>
#include <stdio.h>

int  ch = 'r';

char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int result;

   printf_s( "String to be searched:\n      %s\n", string );
   printf_s( "      %s\n      %s\n\n", fmt1, fmt2 );
   printf_s( "Search char:   %c\n", ch );

   // Search forward.
   pdest = strchr( string, ch );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf_s( "Result:   first %c found at position %d\n",
              ch, result );
   else
      printf_s( "Result:   %c not found\n", ch );

   // Search backward.
   pdest = strrchr( string, ch );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf_s( "Result:   last %c found at position %d\n", ch, result );
   else
      printf_s( "Result:\t%c not found\n", ch );
}
```

```
String to be searched:
      The quick brown dog jumps over the lazy fox
               1         2         3         4         5
      123456789012345678901234567890123456789012345678901

Search char:   r
Result:   first r found at position 12
Result:   last r found at position 30
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strstr, wcsstr, _mbsstr, _mbsstr_l

strcmp, wcscmp, _mbscmp, _mbscmp_l

2/4/2019 • 3 minutes to read • Edit Online

Compare strings.

IMPORTANT

_mbscmp and **_mbscmp_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int strcmp(
   const char *string1,
   const char *string2
);
int wcscmp(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbscmp(
   const unsigned char *string1,
   const unsigned char *string2
);
int _mbscmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

locale
Locale to use.

Return Value

The return value for each of these functions indicates the ordinal relation of *string1* to *string2*.

VALUE	RELATIONSHIP OF STRING1 TO STRING2
< 0	*string1* is less than *string2*
0	*string1* is identical to *string2*
> 0	*string1* is greater than *string2*

On a parameter validation error, **_mbscmp** and **_mbscmp_l** return **_NLSCMPERROR**, which is defined in <string.h> and <mbstring.h>.

Remarks

The **strcmp** function performs an ordinal comparison of *string1* and *string2* and returns a value that indicates their relationship. **wcscmp** and **_mbscmp** are, respectively, wide-character and multibyte-character versions of **strcmp**. **_mbscmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. **_mbscmp_l** has the same behavior, but uses the locale parameter that's passed in instead of the current locale. For more information, see Code Pages. Also, if *string1* or *string2* is a null pointer, **_mbscmp** invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, **_mbscmp** and **_mbscmp_l** return **_NLSCMPERROR** and set **errno** to **EINVAL**. **strcmp** and **wcscmp** do not validate their parameters. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscmp	**strcmp**	**_mbscmp**	**wcscmp**

The **strcmp** functions differ from the **strcoll** functions in that **strcmp** comparisons are ordinal, and are not affected by locale. **strcoll** compares strings lexicographically by using the **LC_COLLATE** category of the current locale. For more information about the **LC_COLLATE** category, see setlocale, _wsetlocale.

In the "C" locale, the order of characters in the character set (ASCII character set) is the same as the lexicographic character order. However, in other locales, the order of characters in the character set may differ from the lexicographic order. For example, in certain European locales, the character 'a' (value 0x61) comes before the character 'ä' (value 0xE4) in the character set, but the character 'ä' comes in front of the character 'a' lexicographically.

In locales for which the character set and the lexicographic character order differ, you can use **strcoll** instead of **strcmp** for lexicographic comparison of strings. Alternatively, you can use **strxfrm** on the original strings, and then use **strcmp** on the resulting strings.

The **strcmp** functions are case-sensitive. **_stricmp**, **_wcsicmp**, and **_mbsicmp** compare strings by first converting them to their lowercase forms. Two strings that contain characters that are located between 'Z' and 'a' in the ASCII table ('[', '\', ']', '^', '_', and '`') compare differently, depending on their case. For example, the two strings "ABCDE" and "ABCD^" compare one way if the comparison is lowercase ("abcde" > "abcd^") and the other way ("ABCDE" < "ABCD^") if the comparison is uppercase.

Requirements

ROUTINE	REQUIRED HEADER
strcmp	<string.h>
wcscmp	<string.h> or <wchar.h>
_mbscmp	<mbstring.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_strcmp.c

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";

int main( void )
{
   char tmp[20];
   int result;

   // Case sensitive
   printf( "Compare strings:\n   %s\n   %s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof (tmp), "less than" );
   else
      strcpy_s( tmp, _countof (tmp), "equal to" );
   printf( "   strcmp:   String 1 is %s string 2\n", tmp );

   // Case insensitive (could use equivalent _stricmp)
   result = _stricmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof (tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof (tmp), "less than" );
   else
      strcpy_s( tmp, _countof (tmp), "equal to" );
   printf( "   _stricmp:  String 1 is %s string 2\n", tmp );
}
```

```
Compare strings:
   The quick brown dog jumps over the lazy fox
   The QUICK brown dog jumps over the lazy fox

   strcmp:   String 1 is greater than string 2
   _stricmp:  String 1 is equal to string 2
```

See also

String Manipulation
memcmp, wmemcmp
_memicmp, _memicmp_l
strcoll Functions
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strcmpi

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _stricmp instead.

strcoll, wcscoll, _mbscoll, _strcoll_l, _wcscoll_l, _mbscoll_l

10/31/2018 • 2 minutes to read • Edit Online

Compares strings by using the current locale or a specified LC_COLLATE conversion-state category.

> **IMPORTANT**
>
> **_mbscoll** and **_mbscoll_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int strcoll(
   const char *string1,
   const char *string2
);
int wcscoll(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbscoll(
   const unsigned char *string1,
   const unsigned char *string2
);
int _strcoll_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int wcscoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbscoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

locale
Locale to use.

Return Value

Each of these functions returns a value indicating the relationship of *string1* to *string2*, as follows.

RETURN VALUE	RELATIONSHIP OF STRING1 TO STRING2
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

Each of these functions returns **_NLSCMPERROR** on an error. To use **_NLSCMPERROR**, include either STRING.H or MBSTRING.H. **wcscoll** can fail if either *string1* or *string2* is **NULL** or contains wide-character codes outside the domain of the collating sequence. When an error occurs, **wcscoll** may set **errno** to **EINVAL**. To check for an error on a call to **wcscoll**, set **errno** to 0 and then check **errno** after calling **wcscoll**.

Remarks

Each of these functions performs a case-sensitive comparison of *string1* and *string2* according to the code page currently in use. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the string comparison.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or if *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

The comparison of the two strings is a locale-dependent operation since each locale has different rules for ordering characters. The versions of these functions without the **_l** suffix use the current thread's locale for this locale-dependent behavior; the versions with the **_l** suffix are identical to the corresponding function without the suffix except that they use the locale passed in as a parameter instead of the current locale. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscoll	**strcoll**	**_mbscoll**	**wcscoll**

Requirements

ROUTINE	REQUIRED HEADER
strcoll	<string.h>
wcscoll	<wchar.h>, <string.h>
_mbscoll, **_mbscoll_l**	<mbstring.h>
_strcoll_l	<string.h>
_wcscoll_l	<wchar.h>, <string.h>

For additional compatibility information, see Compatibility.

See also

Locale
String Manipulation
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strcpy, wcscpy, _mbscpy

3/1/2019 • 2 minutes to read • Edit Online

Copies a string. More secure versions of these functions are available; see strcpy_s, wcscpy_s, _mbscpy_s.

> **IMPORTANT**
>
> **_mbscpy** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strcpy(
   char *strDestination,
   const char *strSource
);
wchar_t *wcscpy(
   wchar_t *strDestination,
   const wchar_t *strSource
);
unsigned char *_mbscpy(
   unsigned char *strDestination,
   const unsigned char *strSource
);
template <size_t size>
char *strcpy(
   char (&strDestination)[size],
   const char *strSource
); // C++ only
template <size_t size>
wchar_t *wcscpy(
   wchar_t (&strDestination)[size],
   const wchar_t *strSource
); // C++ only
template <size_t size>
unsigned char *_mbscpy(
   unsigned char (&strDestination)[size],
   const unsigned char *strSource
); // C++ only
```

Parameters

strDestination
Destination string.

strSource
Null-terminated source string.

Return Value

Each of these functions returns the destination string. No return value is reserved to indicate an error.

Remarks

The **strcpy** function copies *strSource*, including the terminating null character, to the location that's specified by

strDestination. The behavior of **strcpy** is undefined if the source and destination strings overlap.

> **IMPORTANT**
>
> Because **strcpy** does not check for sufficient space in *strDestination* before it copies *strSource*, it is a potential cause of buffer overruns. Therefore, we recommend that you use strcpy_s instead.

wcscpy and **_mbscpy** are, respectively, wide-character and multibyte-character versions of **strcpy**. The arguments and return value of **wcscpy** are wide-character strings; those of **_mbscpy** are multibyte-character strings. These three functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscpy	**strcpy**	**_mbscpy**	**wcscpy**

Requirements

ROUTINE	REQUIRED HEADER
strcpy	<string.h>
wcscpy	<string.h> or <wchar.h>
_mbscpy	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strcpy.c
// compile with: /W3
// This program uses strcpy
// and strcat to build a phrase.

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[80];

   // If you change the previous line to
   //   char string[20];
   // strcpy and strcat will happily overrun the string
   // buffer.  See the examples for strncpy and strncat
   // for safer string handling.

   strcpy( string, "Hello world from " ); // C4996
   // Note: strcpy is deprecated; use strcpy_s instead
   strcat( string, "strcpy " );           // C4996
   // Note: strcat is deprecated; use strcat_s instead
   strcat( string, "and " );              // C4996
   strcat( string, "strcat!" );           // C4996
   printf( "String = %s\n", string );
}
```

```
String = Hello world from strcpy and strcat!
```

See also

String Manipulation
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strcpy_s, wcscpy_s, _mbscpy_s, _mbscpy_s_l

3/1/2019 • 3 minutes to read • Edit Online

Copies a string. These versions of strcpy, wcscpy, _mbscpy have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbscpy_s** and **_mbscpy_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t strcpy_s(
   char *dest,
   rsize_t dest_size,
   const char *src
);
errno_t wcscpy_s(
   wchar_t *dest,
   rsize_t dest_size,
   const wchar_t *src
);
errno_t _mbscpy_s(
   unsigned char *dest,
   rsize_t dest_size,
   const unsigned char *src
);
errno_t _mbscpy_s_l(
   unsigned char *dest,
   rsize_t dest_size,
   const unsigned char *src,
   _locale_t locale
);
```

```
// Template functions are C++ only:
template <size_t size>
errno_t strcpy_s(
   char (&dest)[size],
   const char *src
); // C++ only
template <size_t size>
errno_t wcscpy_s(
   wchar_t (&dest)[size],
   const wchar_t *src
); // C++ only
template <size_t size>
errno_t _mbscpy_s(
   unsigned char (&dest)[size],
   const unsigned char *src
); // C++ only
template <size_t size>
errno_t _mbscpy_s_l(
   unsigned char (&dest)[size],
   const unsigned char *src,
   _locale_t locale
); // C++ only
```

Parameters

dest
Location of the destination string buffer.

dest_size
Size of the destination string buffer in **char** units for narrow and multi-byte functions, and **wchar_t** units for wide functions. This value must be greater than zero and not greater than **RSIZE_MAX**.

src
Null-terminated source string buffer.

locale
Locale to use.

Return Value

Zero if successful; otherwise, an error.

Error Conditions

DEST	*DEST_SIZE*	*SRC*	RETURN VALUE	CONTENTS OF *DEST*
NULL	any	any	**EINVAL**	not modified
any	any	**NULL**	**EINVAL**	*dest*[0] set to 0
any	0, or too small	any	**ERANGE**	*dest*[0] set to 0

Remarks

The **strcpy_s** function copies the contents in the address of *src*, including the terminating null character, to the location that's specified by *dest*. The destination string must be large enough to hold the source string and its terminating null character. The behavior of **strcpy_s** is undefined if the source and destination strings overlap.

wcscpy_s is the wide-character version of **strcpy_s**, and **_mbscpy_s** is the multibyte-character version. The

arguments of **wcscpy_s** are wide-character strings; those of **_mbscpy_s** and **_mbscpy_s_l** are multibyte-character strings. These functions behave identically otherwise. **_mbscpy_s_l** is identical to **_mbscpy_s** except that it uses the locale parameter passed in instead of the current locale. For more information, see Locale.

If *dest* or *src* is a null pointer, or if the destination string size *dest_size* is too small, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL** when *dest* or *src* is a null pointer, and they return **ERANGE** and set **errno** to **ERANGE** when the destination string is too small.

Upon successful execution, the destination string is always null-terminated.

In C++, use of these functions is simplified by template overloads that can infer buffer length automatically so that you don't have to specify a size argument, and they can automatically replace older, less-secure functions with their newer, more secure counterparts. For more information, see Secure Template Overloads.

The debug library versions of these functions first fill the buffer with 0xFE. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscpy_s	**strcpy_s**	**_mbscpy_s**	**wcscpy_s**

Requirements

ROUTINE	REQUIRED HEADER
strcpy_s	<string.h>
wcscpy_s	<string.h> or <wchar.h>
_mbscpy_s	<mbstring.h>

These functions are Microsoft-specific. For additional compatibility information, see Compatibility.

Example

Unlike production quality code, this sample calls the secure string functions without checking for errors:

```
// crt_strcpy_s.c
// Compile by using: cl /W4 crt_strcpy_s.c
// This program uses strcpy_s and strcat_s
// to build a phrase.

#include <string.h>     // for strcpy_s, strcat_s
#include <stdlib.h>     // for _countof
#include <stdio.h>      // for printf
#include <errno.h>      // for return values

int main(void)
{
    char string[80];

    strcpy_s(string, _countof(string), "Hello world from ");
    strcat_s(string, _countof(string), "strcpy_s ");
    strcat_s(string, _countof(string), "and ");
    strcat_s(string, _countof(string), "strcat_s!");

    printf("String = %s\n", string);
}
```

```
String = Hello world from strcpy_s and strcat_s!
```

When building C++ code, the template versions may be easier to use.

```
// crt_wcscpy_s.cpp
// Compile by using: cl /EHsc /W4 crt_wcscpy_s.cpp
// This program uses wcscpy_s and wcscat_s
// to build a phrase.

#include <cstring>  // for wcscpy_s, wcscat_s
#include <cstdlib>  // for _countof
#include <iostream> // for cout, includes <cstdlib>, <cstring>
#include <errno.h>  // for return values

int main(void)
{
    wchar_t string[80];
    // using template versions of wcscpy_s and wcscat_s:
    wcscpy_s(string, L"Hello world from ");
    wcscat_s(string, L"wcscpy_s ");
    wcscat_s(string, L"and ");
    // of course we can supply the size explicitly if we want to:
    wcscat_s(string, _countof(string), L"wcscat_s!");

    std::wcout << L"String = " << string << std::endl;
}
```

```
String = Hello world from wcscpy_s and wcscat_s!
```

See also

String Manipulation
strcat, wcscat, _mbscat, _mbscat_l
strcmp, wcscmp, _mbscmp, _mbscmp_l
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strcspn, wcscspn, _mbscspn, _mbscspn_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the index of the first occurrence in a string, of a character that belongs to a set of characters.

> **IMPORTANT**
>
> **_mbschr** and **_mbschr_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t strcspn(
   const char *str,
   const char *strCharSet
);
size_t wcscspn(
   const wchar_t *str,
   const wchar_t *strCharSet
);
size_t _mbscspn(
   const unsigned char *str,
   const unsigned char *strCharSet
);
size_t _mbscspn_l(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
);
```

Parameters

str
Null-terminated searched string.

strCharSet
Null-terminated character set.

locale
Locale to use.

Return Value

These functions return the index of the first character in *str* that is in *strCharSet*. If none of the characters in *str* is in *strCharSet*, then the return value is the length of *str*.

No return value is reserved to indicate an error.

Remarks

wcscspn and **_mbscspn** are wide-character and multibyte-character versions of **strcspn**. The arguments of **wcscspn** are wide-character strings; those of **_mbscspn** are multibyte-character strings.

_mbscspn validates its parameters. If either *str* or *strCharSet* is a null pointer, the invalid parameter handler is

invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EINVAL**. **strcspn** and **wcscspn** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcscspn	**strcspn**	**_mbscspn**	**wcscspn**
n/a	n/a	**_mbscspn_l**	n/a

Requirements

ROUTINE	REQUIRED HEADER
strcspn	<string.h>
wcscspn	<string.h> or <wchar.h>
_mbscspn, **_mbscspn_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strcspn.c

#include <string.h>
#include <stdio.h>

void test( const char * str, const char * strCharSet )
{
   int pos = strcspn( str, strCharSet );
   printf( "strcspn( \"%s\", \"%s\" ) = %d\n", str, strCharSet, pos );
}

int main( void )
{
   test( "xyzbxz", "abc" );
   test( "xyzbxz", "xyz" );
   test( "xyzbxz", "no match" );
   test( "xyzbxz", "" );
   test( "", "abc" );
   test( "", "" );
}
```

```
strcspn( "xyzbxz", "abc" ) = 3
strcspn( "xyzbxz", "xyz" ) = 0
strcspn( "xyzbxz", "no match" ) = 6
strcspn( "xyzbxz", "" ) = 6
strcspn( "", "abc" ) = 0
strcspn( "", "" ) = 0
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strdate, _wstrdate

10/31/2018 • 2 minutes to read • Edit Online

Copy current system date to a buffer. More secure versions of these functions are available; see _strdate_s, _wstrdate_s.

Syntax

```
char *_strdate(
   char *datestr
);
wchar_t *_wstrdate(
   wchar_t *datestr
);
template <size_t size>
char *_strdate(
   char (&datestr)[size]
); // C++ only
template <size_t size>
wchar_t *_wstrdate(
   wchar_t (&datestr)[size]
); // C++ only
```

Parameters

datestr
A pointer to a buffer containing the formatted date string.

Return Value

Each of these functions returns a pointer to the resulting character string *datestr*.

Remarks

More secure versions of these functions are available; see _strdate_s, _wstrdate_s. It is recommended that the more secure functions be used wherever possible.

The **_strdate** function copies the current system date to the buffer pointed to by *datestr*, formatted **mm**/**dd**/**yy**, where **mm** is two digits representing the month, **dd** is two digits representing the day, and **yy** is the last two digits of the year. For example, the string **12/05/99** represents December 5, 1999. The buffer must be at least 9 bytes long.

If *datestr* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

_wstrdate is a wide-character version of **_strdate**; the argument and return value of **_wstrdate** are wide-character strings. These functions behave identically otherwise.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstrdate	**_strdate**	**_strdate**	**_wstrdate**

Requirements

ROUTINE	REQUIRED HEADER
_strdate	<time.h>
_wstrdate	<time.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// strdate.c
// compile with: /W3
#include <time.h>
#include <stdio.h>
int main()
{
    char tmpbuf[9];

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    printf( "OS date: %s\n", _strdate(tmpbuf) ); // C4996
    // Note: _strdate is deprecated; consider using _strdate_s instead
}
```

```
OS date: 04/25/03
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

_strdate_s, _wstrdate_s

10/31/2018 • 2 minutes to read • Edit Online

Copy the current system date to a buffer. These are versions of _strdate, _wstrdate with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _strdate_s(
   char *buffer,
   size_t numberOfElements
);
errno_t _wstrdate_s(
   wchar_t *buffer,
   size_t numberOfElements
);
template <size_t size>
errno_t _strdate_s(
   char (&buffer)[size]
); // C++ only
template <size_t size>
errno_t _wstrdate_s(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer
A pointer to a buffer which will be filled in with the formatted date string.

numberOfElements
Size of the buffer.

Return Value

Zero if successful. The return value is an error code if there is a failure. Error codes are defined in ERRNO.H; see table below for the exact errors generated by this function. For more information on error codes, see errno.

Error Conditions

BUFFER	*NUMBEROFELEMENTS*	RETURN	CONTENTS OF *BUFFER*
NULL	(any)	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0 < *numberOfElements* < 9	**EINVAL**	Empty string
Not **NULL** (pointing to valid buffer)	*numberOfElements* >= 9	0	Current date formatted as specified in the remarks

Security Issues

Passing in an invalid non **NULL** value for the buffer will result in an access violation if the *numberOfElements* parameter is greater than 9.

Passing values for size that is greater than the actual size of the *buffer* will result in buffer overrun.

Remarks

These functions provide more secure versions of **_strdate** and **_wstrdate**. The **_strdate_s** function copies the current system date to the buffer pointed to by *buffer*, formatted **mm**/**dd**/**yy**, where **mm** is two digits representing the month, **dd** is two digits representing the day, and **yy** is the last two digits of the year. For example, the string **12/05/99** represents December 5, 1999. The buffer must be at least 9 characters long.

_wstrdate_s is a wide-character version of **_strdate_s**; the argument and return value of **_wstrdate_s** are wide-character strings. These functions behave identically otherwise.

If *buffer* is a **NULL** pointer, or if *numberOfElements* is less than 9 characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL** if the buffer is **NULL** or if *numberOfElements* is less than or equal to 0, or set **errno** to **ERANGE** if *numberOfElements* is less than 9.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mapping:

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstrdate_s	**_strdate_s**	**_strdate_s**	**_wstrdate_s**

Requirements

ROUTINE	REQUIRED HEADER
_strdate	<time.h>
_wstrdate	<time.h> or <wchar.h>
_strdate_s	<time.h>

Example

See the example for time.

See also

Time Management
asctime_s, _wasctime_s
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
mktime, _mktime32, _mktime64

time, _time32, _time64
_tzset

_strdec, _wcsdec, _mbsdec, _mbsdec_l

10/31/2018 • 2 minutes to read • Edit Online

Moves a string pointer back one character.

> **IMPORTANT**
> **mbsdec** and **mbsdec_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned char *_strdec(
   const unsigned char *start,
   const unsigned char *current
);
unsigned wchar_t *_wcsdec(
   const unsigned wchar_t *start,
   const unsigned wchar_t *current
);
unsigned char *_mbsdec(
   const unsigned char *start,
   const unsigned char *current
);
unsigned char *_mbsdec_l(
   const unsigned char *start,
   const unsigned char *current,
   _locale_t locale
);
```

Parameters

start
Pointer to any character (or for **_mbsdec** and **_mbsdec_l**, the first byte of any multibyte character) in the source string; *start* must precede *current* in the source string.

current
Pointer to any character (or for **_mbsdec** and **_mbsdec_l**, the first byte of any multibyte character) in the source string; *current* must follow *start* in the source string.

locale
Locale to use.

Return Value

_mbsdec, **_mbsdec_l**, **_strdec**, and **_wcsdec** each return a pointer to the character that immediately precedes *current*; **_mbsdec** returns **NULL** if the value of *start* is greater than or equal to that of *current*. **_tcsdec** maps to one of these functions and its return value depends on the mapping.

Remarks

The **_mbsdec** and **_mbsdec_l** functions return a pointer to the first byte of the multibyte character that immediately precedes *current* in the string that contains *start*.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. **_mbsdec** recognizes multibyte-character sequences according to the locale that's currently in use, while **_mbsdec_l** is identical except that it instead uses the locale parameter that's passed in. For more information, see Locale.

If *start* or *current* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

> **IMPORTANT**
> These functions might be vulnerable to buffer overrun threats. Buffer overruns can be used for system attacks because they can cause an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsdec	**_strdec**	**_mbsdec**	**_wcsdec**

_strdec and **_wcsdec** are single-byte-character and wide-character versions of **_mbsdec** and **_mbsdec_l**. **_strdec** and **_wcsdec** are provided only for this mapping and should not be used otherwise.

For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADER
_mbsdec	<mbstring.h>	<mbctype.h>
_mbsdec_l	<mbstring.h>	<mbctype.h>
_strdec	<tchar.h>	
_wcsdec	<tchar.h>	

For more compatibility information, see Compatibility.

Example

The following example shows a use of **_tcsdec**.

```
// crt_tcsdec.cpp
// Compile by using: cl /EHsc crt_tcsdec.cpp
#include <iostream>
#include <tchar.h>
using namespace std;

int main()
{
   const TCHAR *str = _T("12345");
   cout << "str: " << str << endl;

   const TCHAR *str2;
   str2 = str + 2;
   cout << "str2: " << str2 << endl;

   TCHAR *answer;
   answer = _tcsdec( str, str2 );
   cout << "answer: " << answer << endl;

   return (0);
}
```

The following example shows a use of **_mbsdec**.

```
// crt_mbsdec.cpp
// Compile by using: cl /EHsc crt_mbsdec.c
#include <iostream>
#include <mbstring.h>
using namespace std;

int main()
{
   char *str = "12345";
   cout << "str: " << str << endl;

   char *str2;
   str2 = str + 2;
   cout << "str2: " << str2 << endl;

   unsigned char *answer;
   answer = _mbsdec( reinterpret_cast<unsigned char *>( str ), reinterpret_cast<unsigned char *>( str2 ));

   cout << "answer: " << answer << endl;

   return (0);
}
```

See also

String Manipulation
_strinc, _wcsinc, _mbsinc, _mbsinc_l
_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l
_strninc, _wcsninc, _mbsninc, _mbsninc_l

strdup, wcsdup

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strdup, _wcsdup, _mbsdup instead.

_strdup, _wcsdup, _mbsdup

10/31/2018 • 2 minutes to read • Edit Online

Duplicates strings.

> **IMPORTANT**
> **_mbsdup** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strdup(
   const char *strSource
);
wchar_t *_wcsdup(
   const wchar_t *strSource
);
unsigned char *_mbsdup(
   const unsigned char *strSource
);
```

Parameters

strSource
Null-terminated source string.

Return Value

Each of these functions returns a pointer to the storage location for the copied string or **NULL** if storage cannot be allocated.

Remarks

The **_strdup** function calls malloc to allocate storage space for a copy of *strSource* and then copies *strSource* to the allocated space.

_wcsdup and **_mbsdup** are wide-character and multibyte-character versions of **_strdup**. The arguments and return value of **_wcsdup** are wide-character strings; those of **_mbsdup** are multibyte-character strings. These three functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsdup	**_strdup**	**_mbsdup**	**_wcsdup**

Because **_strdup** calls **malloc** to allocate storage space for the copy of *strSource*, it is good practice always to release this memory by calling the free routine on the pointer that's returned by the call to **_strdup**.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, **_strdup** and **_wcsdup** are replaced by calls to **_strdup_dbg** and **_wcsdup_dbg** to allow for debugging memory allocations. For more information, see

_strdup_dbg, _wcsdup_dbg.

Requirements

ROUTINE	REQUIRED HEADER
_strdup	<string.h>
_wcsdup	<string.h> or <wchar.h>
_mbsdup	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strdup.c

#include <string.h>
#include <stdio.h>

int main( void )
{
   char buffer[] = "This is the buffer text";
   char *newstring;
   printf( "Original: %s\n", buffer );
   newstring = _strdup( buffer );
   printf( "Copy:     %s\n", newstring );
   free( newstring );
}
```

```
Original: This is the buffer text
Copy:     This is the buffer text
```

See also

String Manipulation
memset, wmemset
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strdup_dbg, _wcsdup_dbg

10/31/2018 • 2 minutes to read • Edit Online

Versions of _strdup and _wcsdup that use the debug version of **malloc**.

Syntax

```
char *_strdup_dbg(
   const char *strSource,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wcsdup_dbg(
   const wchar_t *strSource,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

strSource
Null-terminated source string.

blockType
Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to name of source file that requested allocation operation or **NULL**.

linenumber
Line number in source file where allocation operation was requested or **NULL**.

Return Value

Each of these functions returns a pointer to the storage location for the copied string or **NULL** if storage cannot be allocated.

Remarks

The **_strdup_dbg** and **_wcsdup_dbg** functions are identical to **_strdup** and **_wcsdup** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc**, **_malloc_dbg**, to allocate memory for the duplicated string. For information on the debugging features of **_malloc_dbg**, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_strdup** and **_wcsdup** are remapped to **_strdup_dbg** and **_wcsdup_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information on block types, see Types of blocks on the debug heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsdup_dbg	**_strdup_dbg**	**_mbsdup**	**_wcsdup_dbg**

Requirements

ROUTINE	REQUIRED HEADER
_strdup_dbg, **_wcsdup_dbg**	<crtdbg.h>

For additional compatibility information, see Compatibility.

Libraries

All debug versions of the C run-time libraries.

See also

String Manipulation
_strdup, _wcsdup, _mbsdup
Debug Versions of Heap Allocation Functions

strerror, _strerror, _wcserror, __wcserror

10/31/2018 • 2 minutes to read • Edit Online

Gets a system error message string (**strerror**, **_wcserror**) or formats a user-supplied error message string (**_strerror**, **__wcserror**). More secure versions of these functions are available; see strerror_s, _strerror_s, _wcserror_s, __wcserror_s.

Syntax

```
char *strerror(
   int errnum
);
char *_strerror(
   const char *strErrMsg
);
wchar_t * _wcserror(
   int errnum
);
wchar_t * __wcserror(
   const wchar_t *strErrMsg
);
```

Parameters

errnum
Error number.

strErrMsg
User-supplied message.

Return Value

All of these functions return a pointer to the error-message string. Subsequent calls can overwrite the string.

Remarks

The **strerror** function maps *errnum* to an error-message string and returns a pointer to the string. Neither **strerror** nor **_strerror** actually prints the message: For that, you have to call an output function such as fprintf:

```
if (( _access( "datafile",2 )) == -1 )
   fprintf( stderr, _strerror(NULL) );
```

If *strErrMsg* is passed as **NULL**, **_strerror** returns a pointer to a string that contains the system error message for the last library call that produced an error. The error-message string is terminated by the newline character ('\n'). If *strErrMsg* is not equal to **NULL**, then **_strerror** returns a pointer to a string that contains (in order) your string message, a colon, a space, the system error message for the last library call that produces an error, and a newline character. Your string message can be, at most, 94 characters long.

The actual error number for **_strerror** is stored in the variable errno. To produce accurate results, call **_strerror** immediately after a library routine returns with an error. Otherwise, subsequent calls to **strerror** or **_strerror** can overwrite the **errno** value.

_wcserror and **__wcserror** are wide-character versions of **strerror** and **_strerror**, respectively.

_strerror, **_wcserror**, and **__wcserror** are not part of the ANSI definition; they are Microsoft extensions and we recommend that you do not use them where you want portable code. For ANSI compatibility, use **strerror** instead.

To get error strings, we recommend **strerror** or **_wcserror** instead of the deprecated macros _sys_errlist and _sys_nerr and the deprecated internal functions **__sys_errlist** and **__sys_nerr**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcserror	**strerror**	**strerror**	**_wcserror**

Requirements

ROUTINE	REQUIRED HEADER
strerror	<string.h>
_strerror	<string.h>
_wcserror, **__wcserror**	<string.h>

For additional compatibility information, see Compatibility.

Example

See the example for perror.

See also

String Manipulation
clearerr
ferror
perror, _wperror

strerror_s, _strerror_s, _wcserror_s, __wcserror_s

10/31/2018 • 3 minutes to read • Edit Online

Get a system error message (**strerror_s**, **_wcserror_s**) or print a user-supplied error message (**_strerror_s**, **__wcserror_s**). These are versions of strerror, _strerror, _wcserror, __wcserror with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t strerror_s(
   char *buffer,
   size_t numberOfElements,
   int errnum
);
errno_t _strerror_s(
   char *buffer,
   size_t numberOfElements,
   const char *strErrMsg
);
errno_t _wcserror_s(
   wchar_t *buffer,
   size_t numberOfElements,
   int errnum
);
errno_t __wcserror_s(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *strErrMsg
);
template <size_t size>
errno_t strerror_s(
   char (&buffer)[size],
   int errnum
); // C++ only
template <size_t size>
errno_t _strerror_s(
   char (&buffer)[size],
   const char *strErrMsg
); // C++ only
template <size_t size>
errno_t _wcserror_s(
   wchar_t (&buffer)[size],
   int errnum
); // C++ only
template <size_t size>
errno_t __wcserror_s(
   wchar_t (&buffer)[size],
   const wchar_t *strErrMsg
); // C++ only
```

Parameters

buffer
Buffer to hold error string.

numberOfElements
Size of buffer.

errnum

Error number.

strErrMsg
User-supplied message.

Return Value

Zero if successful, an error code on failure.

Error Condtions

BUFFER	*NUMBEROFELEMENTS*	*STRERRMSG*	CONTENTS OF *BUFFER*
NULL	any	any	n/a
any	0	any	not modified

Remarks

The **strerror_s** function maps *errnum* to an error-message string, returning the string in *buffer*. **_strerror_s** doesn't take the error number; it uses the current value of **errno** to determine the appropriate message. Neither **strerror_s** nor **_strerror_s** actually prints the message: For that, you need to call an output function such as fprintf:

```
if (( _access( "datafile",2 )) == -1 )
{
   _strerror_s(buffer, 80);
   fprintf( stderr, buffer );
}
```

If *strErrMsg* is **NULL**, **_strerror_s** returns a string in *buffer* containing the system error message for the last library call that produced an error. The error-message string is terminated by the newline character ('\n'). If *strErrMsg* is not equal to **NULL**, then **_strerror_s** returns a string in *buffer* containing (in order) your string message, a colon, a space, the system error message for the last library call producing an error, and a newline character. Your string message can be, at most, 94 characters long.

These functions truncate the error message if its length exceeds *numberOfElements* -1. The resulting string in *buffer* is always null-terminated.

The actual error number for **_strerror_s** is stored in the variable errno. The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. **_strerror_s** accesses the appropriate error message by using the **errno** value as an index to the variable **_sys_errlist**. The value of the variable _sys_nerr is defined as the maximum number of elements in the **_sys_errlist** array. To produce accurate results, call **_strerror_s** immediately after a library routine returns with an error. Otherwise, subsequent calls to **strerror_s** or **_strerror_s** can overwrite the **errno** value.

_wcserror_s and **__wcserror_s** are wide-character versions of **strerror_s** and **_strerror_s**, respectively.

These functions validate their parameters. If buffer is **NULL** or if the size parameter is 0, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**.

_strerror_s, **_wcserror_s**, and **__wcserror_s** are not part of the ANSI definition but are instead Microsoft extensions to it. Do not use them where portability is desired; for ANSI compatibility, use **strerror_s** instead.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length

automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcserror_s	**strerror_s**	**strerror_s**	**_wcserror_s**

Requirements

ROUTINE	REQUIRED HEADER
strerror_s, **_strerror_s**	<string.h>
_wcserror_s, **__wcserror_s**	<string.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for perror.

See also

String Manipulation
clearerr
ferror
perror, _wperror

strftime, wcsftime, _strftime_l, _wcsftime_l

10/31/2018 • 5 minutes to read • Edit Online

Format a time string.

Syntax

```
size_t strftime(
   char *strDest,
   size_t maxsize,
   const char *format,
   const struct tm *timeptr
);
size_t _strftime_l(
   char *strDest,
   size_t maxsize,
   const char *format,
   const struct tm *timeptr,
   _locale_t locale
);
size_t wcsftime(
   wchar_t *strDest,
   size_t maxsize,
   const wchar_t *format,
   const struct tm *timeptr
);
size_t _wcsftime_l(
   wchar_t *strDest,
   size_t maxsize,
   const wchar_t *format,
   const struct tm *timeptr,
   _locale_t locale
);
```

Parameters

strDest
Output string.

maxsize
Size of the *strDest* buffer, measured in characters (**char** or **wchar_t**).

format
Format-control string.

timeptr
tm data structure.

locale
The locale to use.

Return Value

strftime returns the number of characters placed in *strDest* and **wcsftime** returns the corresponding number of wide characters.

If the total number of characters, including the terminating null, is more than *maxsize*, both **strftime** and

wcsftime return 0 and the contents of *strDest* are indeterminate.

The number of characters in *strDest* is equal to the number of literal characters in *format* as well as any characters that may be added to *format* via formatting codes. The terminating null of a string is not counted in the return value.

Remarks

The **strftime** and **wcsftime** functions format the **tm** time value in *timeptr* according to the supplied *format* argument and store the result in the buffer *strDest*. At most, *maxsize* characters are placed in the string. For a description of the fields in the *timeptr* structure, see asctime. **wcsftime** is the wide-character equivalent of **strftime**; its string-pointer argument points to a wide-character string. These functions behave identically otherwise.

This function validates its parameters. If *strDest*, *format*, or *timeptr* is a null pointer, or if the **tm** data structure addressed by *timeptr* is invalid (for example, if it contains out of range values for the time or date), or if the *format* string contains an invalid formatting code, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the function returns 0 and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsftime	**strftime**	**strftime**	**wcsftime**

The *format* argument consists of one or more codes; as in **printf**, the formatting codes are preceded by a percent sign (**%**). Characters that do not begin with **%** are copied unchanged to *strDest*. The **LC_TIME** category of the current locale affects the output formatting of **strftime**. (For more information on **LC_TIME**, see setlocale.) The **strftime** and **wcsftime** functions use the currently set locale. The **_strftime_l** and **_wcsftime_l** versions of these functions are identical except that they take the locale as a parameter and use that instead of the currently set locale. For more information, see Locale.

The **strftime** functions support these formatting codes:

Code	Replacement string
%a	Abbreviated weekday name in the locale
%A	Full weekday name in the locale
%b	Abbreviated month name in the locale
%B	Full month name in the locale
%c	Date and time representation appropriate for locale
%C	The year divided by 100 and truncated to an integer, as a decimal number (00–99)
%d	Day of month as a decimal number (01 - 31)
%D	Equivalent to **%m/%d/%y**

%e	Day of month as a decimal number (1 - 31), where single digits are preceded by a space
%F	Equivalent to **%Y-%m-%d**
%g	The last 2 digits of the ISO 8601 week-based year as a decimal number (00 - 99)
%G	The ISO 8601 week-based year as a decimal number
%h	Abbreviated month name (equivalent to **%b**)
%H	Hour in 24-hour format (00 - 23)
%I	Hour in 12-hour format (01 - 12)
%j	Day of the year as a decimal number (001 - 366)
%m	Month as a decimal number (01 - 12)
%M	Minute as a decimal number (00 - 59)
%n	A newline character (**\n**)
%p	The locale's A.M./P.M. indicator for 12-hour clock
%r	The locale's 12-hour clock time
%R	Equivalent to **%H:%M**
%S	Second as a decimal number (00 - 59)
%t	A horizontal tab character (**\t**)
%T	Equivalent to **%H:%M:%S**, the ISO 8601 time format
%u	ISO 8601 weekday as a decimal number (1 - 7; Monday is 1)
%U	Week number of the year as a decimal number (00 - 53), where the first Sunday is the first day of week 1
%V	ISO 8601 week number as a decimal number (00 - 53)
%w	Weekday as a decimal number (0 - 6; Sunday is 0)
%W	Week number of the year as a decimal number (00 - 53), where the first Monday is the first day of week 1
%x	Date representation for the locale
%X	Time representation for the locale

%y	Year without century, as decimal number (00 - 99)
%Y	Year with century, as decimal number
%z	The offset from UTC in ISO 8601 format; no characters if time zone is unknown
%Z	Either the locale's time-zone name or time zone abbreviation, depending on registry settings; no characters if time zone is unknown
%%	Percent sign

As in the **printf** function, the **#** flag may prefix any formatting code. In that case, the meaning of the format code is changed as follows.

FORMAT CODE	MEANING
%#a, **%#A**, **%#b**, **%#B**, **%#g**, **%#G**, **%#h**, **%#n**, **%#p**, **%#t**, **%#u**, **%#w**, **%#X**, **%#z**, **%#Z**, **%#%**	**#** flag is ignored.
%#c	Long date and time representation, appropriate for the locale. For example: "Tuesday, March 14, 1995, 12:41:29".
%#x	Long date representation, appropriate to the locale. For example: "Tuesday, March 14, 1995".
%#d, **%#D**, **%#e**, **%#F**, **%#H**, **%#I**, **%#j**, **%#m**, **%#M**, **%#r**, **%#R**, **%#S**, **%#T**, **%#U**, **%#V**, **%#W**, **%#y**, **%#Y**	Remove leading zeros or spaces (if any).

The ISO 8601 week and week-based year produced by **%V**, **%g**, and **%G**, uses a week that begins on Monday, where week 1 is the week that contains January 4th, which is the first week that includes at least four days of the year. If the first Monday of the year is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. For those days, **%V** is replaced by 53, and both **%g** and **%G** are replaced by the digits of the preceding year.

Requirements

ROUTINE	REQUIRED HEADER
strftime	<time.h>
wcsftime	<time.h> or <wchar.h>
_strftime_l	<time.h>
_wcsftime_l	<time.h> or <wchar.h>

The **_strftime_l** and **_wcsftime_l** functions are Microsoft-specific. For additional compatibility information, see Compatibility.

Example

See the example for time.

See also

Locale
Time Management
String Manipulation
localeconv
setlocale, _wsetlocale
strcoll Functions
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

stricmp, wcsicmp

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l instead.

_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l

10/31/2018 • 4 minutes to read • Edit Online

Performs a case-insensitive comparison of strings.

> **IMPORTANT**
>
> **_mbsicmp** and **_mbsicmp_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _stricmp(
   const char *string1,
   const char *string2
);
int _wcsicmp(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbsicmp(
   const unsigned char *string1,
   const unsigned char *string2
);
int _stricmp_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int _wcsicmp_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbsicmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

locale
Locale to use.

Return Value

The return value indicates the relation of *string1* to *string2* as follows.

RETURN VALUE	DESCRIPTION
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

On an error, **_mbsicmp** returns **_NLSCMPERROR**, which is defined in <string.h> and <mbstring.h>.

Remarks

The **_stricmp** function ordinally compares *string1* and *string2* after converting each character to lowercase, and returns a value indicating their relationship. **_stricmp** differs from **_stricoll** in that the **_stricmp** comparison is only affected by **LC_CTYPE**, which determines which characters are upper and lowercase. The **_stricoll** function compares strings according to both the **LC_CTYPE** and **LC_COLLATE** categories of the locale, which includes both the case and the collation order. For more information about the **LC_COLLATE** category, see setlocale and Locale Categories. The versions of these functions without the **_l** suffix use the current locale for locale-dependent behavior. The versions with the suffix are identical except that they use the locale passed in instead. If the locale has not been set, the C locale is used. For more information, see Locale.

> **NOTE**
> **_stricmp** is equivalent to **_strcmpi**. They can be used interchangeably but **_stricmp** is the preferred standard.

The **_strcmpi** function is equivalent to **_stricmp** and is provided for backward compatibility only.

Because **_stricmp** does lowercase comparisons, it may result in unexpected behavior.

To illustrate when case conversion by **_stricmp** affects the outcome of a comparison, assume that you have the two strings JOHNSTON and JOHN_HENRY. The string JOHN_HENRY will be considered less than JOHNSTON because the "_" has a lower ASCII value than a lowercase S. In fact, any character that has an ASCII value between 91 and 96 will be considered less than any letter.

If the strcmp function is used instead of **_stricmp**, JOHN_HENRY will be greater than JOHNSTON.

_wcsicmp and **_mbsicmp** are wide-character and multibyte-character versions of **_stricmp**. The arguments and return value of **_wcsicmp** are wide-character strings; those of **_mbsicmp** are multibyte-character strings. **_mbsicmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. For more information, see Code Pages. These three functions behave identically otherwise.

_wcsicmp and **wcscmp** behave identically except that **wcscmp** does not convert its arguments to lowercase before comparing them. **_mbsicmp** and **_mbscmp** behave identically except that **_mbscmp** does not convert its arguments to lowercase before comparing them.

You will need to call setlocale for **_wcsicmp** to work with Latin 1 characters. The C locale is in effect by default, so, for example, ä will not compare equal to Ä. Call **setlocale** with any locale other than the C locale before the call to **_wcsicmp**. The following sample demonstrates how **_wcsicmp** is sensitive to the locale:

```
// crt_stricmp_locale.c
#include <string.h>
#include <stdio.h>
#include <locale.h>

int main() {
   setlocale(LC_ALL,"C");   // in effect by default
   printf("\n%d",_wcsicmp(L"ä", L"Ä"));   // compare fails
   setlocale(LC_ALL,"");
   printf("\n%d",_wcsicmp(L"ä", L"Ä"));   // compare succeeds
}
```

An alternative is to call _create_locale, _wcreate_locale and pass the returned locale object as a parameter to **_wcsicmp_l**.

All of these functions validate their parameters. If either *string1* or *string2* are null pointers, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsicmp	**_stricmp**	**_mbsicmp**	**_wcsicmp**

Requirements

ROUTINE	REQUIRED HEADER
_stricmp, **_stricmp_l**	<string.h>
_wcsicmp, **_wcsicmp_l**	<string.h> or <wchar.h>
_mbsicmp, **_mbsicmp_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_stricmp.c

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";

int main( void )
{
   char tmp[20];
   int result;

   // Case sensitive
   printf( "Compare strings:\n   %s\n   %s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof(tmp), "less than" );
   else
      strcpy_s( tmp, _countof(tmp), "equal to" );
   printf( "   strcmp:   String 1 is %s string 2\n", tmp );

   // Case insensitive (could use equivalent _stricmp)
   result = _stricmp( string1, string2 );
   if( result > 0 )
      strcpy_s( tmp, _countof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, _countof(tmp), "less than" );
   else
      strcpy_s( tmp, _countof(tmp), "equal to" );
   printf( "   _stricmp:  String 1 is %s string 2\n", tmp );
}
```

```
Compare strings:
   The quick brown dog jumps over the lazy fox
   The QUICK brown dog jumps over the lazy fox

   strcmp:   String 1 is greater than string 2
   _stricmp:  String 1 is equal to string 2
```

See also

String Manipulation
memcmp, wmemcmp
_memicmp, _memicmp_l
strcmp, wcscmp, _mbscmp
strcoll Functions
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_stricoll, _wcsicoll, _mbsicoll, _stricoll_l, _wcsicoll_l, _mbsicoll_l

10/31/2018 • 2 minutes to read • Edit Online

Compares strings by using locale-specific information.

> **IMPORTANT**
>
> **_mbsicoll** and **_mbsicoll_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _stricoll(
   const char *string1,
   const char *string2
);
int _wcsicoll(
   const wchar_t *string1,
   const wchar_t *string2
);
int _mbsicoll(
   const unsigned char *string1,
   const unsigned char *string2
);
int _stricoll_l(
   const char *string1,
   const char *string2,
   _locale_t locale
);
int _wcsicoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   _locale_t locale
);
int _mbsicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

locale
The locale to use.

Return Value

Each of these functions returns a value indicating the relationship of *string1* to *string2*, as follows.

RETURN VALUE	RELATIONSHIP OF STRING1 TO STRING2
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*
_NLSCMPERROR	An error occurred.

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either <string.h> or <mbstring.h>. **_wcsicoll** can fail if either *string1* or *string2* contains wide-character codes outside the domain of the collating sequence. When an error occurs, **_wcsicoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsicoll**, set **errno** to 0 and then check **errno** after calling **_wcsicoll**.

Remarks

Each of these functions performs a case-insensitive comparison of *string1* and *string2* according to the code page currently in use. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the current code page and this difference is of interest for the string comparison.

_stricmp differs from **_stricoll** in that the **_stricmp** comparison is affected by **LC_CTYPE**, whereas the **_stricoll** comparison is according to the **LC_CTYPE** and **LC_COLLATE** categories of the locale. For more information on the **LC_COLLATE** category, see setlocale and Locale Categories. The versions of these functions without the **_l** suffix use the current locale; the versions with the **_l** suffix are identical except that they use the locale passed in instead. For more information, see Locale.

All of these functions validate their parameters. If either *string1* or *string2* are **NULL** pointers, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsicoll	**_stricoll**	**_mbsicoll**	**_wcsicoll**

Requirements

ROUTINE	REQUIRED HEADER
_stricoll, **_stricoll_l**	<string.h>
_wcsicoll, **_wcsicoll_l**	<wchar.h>, <string.h>
_mbsicoll, **_mbsicoll_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

See also

Locale

String Manipulation
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

_strinc, _wcsinc, _mbsinc, _mbsinc_l

11/9/2018 • 2 minutes to read • Edit Online

Advances a string pointer by one character.

> **IMPORTANT**
> **_mbsinc** and **_mbsinc_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strinc(
   const char *current,
   _locale_t locale
);
wchar_t *_wcsinc(
   const wchar_t *current,
   _locale_t locale
);
unsigned char *_mbsinc(
   const unsigned char *current
);
unsigned char *_mbsinc_l(
   const unsigned char *current,
   _locale_t locale
);
```

Parameters

current
Character pointer.

locale
Locale to use.

Return Value

Each of these routines returns a pointer to the character that immediately follows *current*.

Remarks

The **_mbsinc** function returns a pointer to the first byte of the multibyte character that immediately follows *current*. **_mbsinc** recognizes multibyte-character sequences according to the multibyte code page that's currently in use; **_mbsinc_l** is identical except that it instead uses the locale parameter that's passed in. For more information, see Locale.

The generic-text function **_tcsinc**, defined in Tchar.h, maps to **_mbsinc** if **_MBCS** has been defined, or to **_wcsinc** if **_UNICODE** has been defined. Otherwise, **_tcsinc** maps to **_strinc**. **_strinc** and **_wcsinc** are single-byte-character and wide-character versions of **_mbsinc**. **_strinc** and **_wcsinc** are provided only for this mapping and should not be used otherwise. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

If *current* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, this function returns **EINVAL** and sets **errno** to **EINVAL**.

> **IMPORTANT**
>
> These functions might be vulnerable to buffer overrun threats. Buffer overruns can be used for system attacks because they can cause an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Requirements

ROUTINE	REQUIRED HEADER
_mbsinc	<mbstring.h>
_mbsinc_l	<mbstring.h>
_strinc	<tchar.h>
_wcsinc	<tchar.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
_strdec, _wcsdec, _mbsdec, _mbsdec_l
_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l
_strninc, _wcsninc, _mbsninc, _mbsninc_l

strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l

3/1/2019 • 3 minutes to read • Edit Online

Gets the length of a string, by using the current locale or a specified locale. More secure versions of these functions are available; see strnlen, strnlen_s, wcsnlen, wcsnlen_s, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

> **IMPORTANT**
>
> **_mbslen**, **_mbslen_l**, **_mbstrlen**, and **_mbstrlen_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t strlen(
   const char *str
);
size_t wcslen(
   const wchar_t *str
);
size_t _mbslen(
   const unsigned char *str
);
size_t _mbslen_l(
   const unsigned char *str,
   _locale_t locale
);
size_t _mbstrlen(
   const char *str
);
size_t _mbstrlen_l(
   const char *str,
   _locale_t locale
);
```

Parameters

str
Null-terminated string.

locale
Locale to use.

Return Value

Each of these functions returns the number of characters in *str*, excluding the terminal null. No return value is reserved to indicate an error, except for **_mbstrlen** and **_mbstrlen_l**, which return `((size_t)(-1))` if the string contains an invalid multibyte character.

Remarks

strlen interprets the string as a single-byte character string, so its return value is always equal to the number of

bytes, even if the string contains multibyte characters. **wcslen** is a wide-character version of **strlen**; the argument of **wcslen** is a wide-character string and the count of characters is in wide (two-byte) characters. **wcslen** and **strlen** behave identically otherwise.

Security Note These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcslen	**strlen**	**strlen**	**wcslen**
_tcsclen	**strlen**	**_mbslen**	**wcslen**
_tcsclen_l	**strlen**	**_mbslen_l**	**wcslen**

_mbslen and **_mbslen_l** return the number of multibyte characters in a multibyte-character string but they do not test for multibyte-character validity. **_mbstrlen** and **_mbstrlen_l** test for multibyte-character validity and recognize multibyte-character sequences. If the string passed to **_mbstrlen** or **_mbstrlen_l** contains an invalid multibyte character for the code page, the function returns -1 and sets **errno** to **EILSEQ**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
strlen	<string.h>
wcslen	<string.h> or <wchar.h>
_mbslen, **_mbslen_l**	<mbstring.h>
_mbstrlen, **_mbstrlen_l**	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strlen.c
// Determine the length of a string. For the multi-byte character
// example to work correctly, the Japanese language support for
// non-Unicode programs must be enabled by the operating system.

#include <string.h>
#include <locale.h>

int main()
{
   char* str1 = "Count.";
   wchar_t* wstr1 = L"Count.";
   char * mbstr1;
   char * locale_string;

   // strlen gives the length of single-byte character string
   printf("Length of '%s' : %d\n", str1, strlen(str1) );

   // wstrlen gives the length of a wide character string
   wprintf(L"Length of '%s' : %d\n", wstr1, wcslen(wstr1) );

   // A multibyte string: [A] [B] [C] [katakana A] [D] [\0]
   // in Code Page 932. For this example to work correctly,
   // the Japanese language support must be enabled by the
   // operating system.
   mbstr1 = "ABC" "\x83\x40" "D";

   locale_string = setlocale(LC_CTYPE, "Japanese_Japan");

   if (locale_string == NULL)
   {
      printf("Japanese locale not enabled. Exiting.\n");
      exit(1);
   }
   else
   {
      printf("Locale set to %s\n", locale_string);
   }

   // _mbslen will recognize the Japanese multibyte character if the
   // current locale used by the operating system is Japanese
   printf("Length of '%s' : %d\n", mbstr1, _mbslen(mbstr1) );

   // _mbstrlen will recognize the Japanese multibyte character
   // since the CRT locale is set to Japanese even if the OS locale
   // isnot.
   printf("Length of '%s' : %d\n", mbstr1, _mbstrlen(mbstr1) );
   printf("Bytes in '%s' : %d\n", mbstr1, strlen(mbstr1) );

}
```

```
Length of 'Count.' : 6
Length of 'Count.' : 6
Length of 'ABCｱD' : 5
Length of 'ABCｱD' : 5
Bytes in 'ABCｱD' : 6
```

See also

String Manipulation
Interpretation of Multibyte-Character Sequences
Locale
setlocale, _wsetlocale

strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcoll Functions
strcpy, wcscpy, _mbscpy
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strlwr, wcslwr

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l or security-enhanced _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l instead.

_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

3/1/2019 • 2 minutes to read • Edit Online

Converts a string to lowercase. More secure versions of these functions are available; see _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l.

> **IMPORTANT**
>
> **_mbslwr** and **_mbslwr_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strlwr(
   char * str
);
wchar_t *_wcslwr(
   wchar_t * str
);
unsigned char *_mbslwr(
   unsigned char * str
);
char *_strlwr_l(
   char * str,
   _locale_t locale
);
wchar_t *_wcslwr_l(
   wchar_t * str,
   _locale_t locale
);
unsigned char *_mbslwr_l(
   unsigned char * str,
   _locale_t locale
);
template <size_t size>
char *_strlwr(
   char (&str)[size]
); // C++ only
template <size_t size>
wchar_t *_wcslwr(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
unsigned char *_mbslwr(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
char *_strlwr_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
wchar_t *_wcslwr_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
unsigned char *_mbslwr_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str
Null-terminated string to convert to lowercase.

locale
The locale to use.

Return Value

Each of these functions returns a pointer to the converted string. Because the modification is done in place, the pointer returned is the same as the pointer passed as the input argument. No return value is reserved to indicate an error.

Remarks

The **_strlwr** function converts any uppercase letters in *str* to lowercase as determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale for their locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale passed in instead. For more information, see Locale.

The **_wcslwr** and **_mbslwr** functions are wide-character and multibyte-character versions of **_strlwr**. The argument and return value of **_wcslwr** are wide-character strings; those of **_mbslwr** are multibyte-character strings. These three functions behave identically otherwise.

If *str* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return the original string and set **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcslwr	**_strlwr**	**_mbslwr**	**_wcslwr**
_tcslwr_l	**_strlwr_l**	**_mbslwr_l**	**_wcslwr_l**

Requirements

ROUTINE	REQUIRED HEADER
_strlwr, **_strlwr_l**	<string.h>
_wcslwr, **_wcslwr_l**	<string.h> or <wchar.h>
_mbslwr, **_mbslwr_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strlwr.c
// compile with: /W3
// This program uses _strlwr and _strupr to create
// uppercase and lowercase copies of a mixed-case string.
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[100] = "The String to End All Strings!";
   char * copy1 = _strdup( string ); // make two copies
   char * copy2 = _strdup( string );

   _strlwr( copy1 ); // C4996
   // Note: _strlwr is deprecated; consider using _strlwr_s instead
   _strupr( copy2 ); // C4996
   // Note: _strupr is deprecated; consider using _strupr_s instead

   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );

   free( copy1 );
   free( copy2 );
}
```

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

See also

String Manipulation

Locale

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a string to lowercase, by using the current locale or a locale object that's passed in. These versions of _strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbslwr_s** and **_mbslwr_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _strlwr_s(
   char *str,
   size_t numberOfElements
);
errno_t _strlwr_s_l(
   char *str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _mbslwr_s(
   unsigned char *str,
   size_t numberOfElements
);
errno_t _mbslwr_s_l(
   unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _wcslwr_s(
   wchar_t *str,
   size_t numberOfElements
);
errno_t _wcslwr_s_l(
   wchar_t *str,
   size_t numberOfElements,
   _locale_t locale
);
template <size_t size>
errno_t _strlwr_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _strlwr_s_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbslwr_s(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
errno_t _mbslwr_s_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _wcslwr_s(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
errno_t _wcslwr_s_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str
Null-terminated string to convert to lowercase.

numberOfElements
Size of the buffer.

locale
The locale to use.

Return Value

Zero if successful; a non-zero error code on failure.

These functions validate their parameters. If *str* is not a valid null-terminated string, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**. If *numberOfElements* is less than the length of the string, the functions also return **EINVAL** and set **errno** to **EINVAL**.

Remarks

The **_strlwr_s** function converts, in place, any uppercase letters in *str* to lowercase. **_mbslwr_s** is a multi-byte character version of **_strlwr_s**. **_wcslwr_s** is a wide-character version of **_strlwr_s**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcslwr_s	**_strlwr_s**	**_mbslwr_s**	**_wcslwr_s**
_tcslwr_s_l	**_strlwr_s_l**	**_mbslwr_s_l**	**_wcslwr_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_strlwr_s, **_strlwr_s_l**	<string.h>
_mbslwr_s, **_mbslwr_s_l**	<mbstring.h>
_wcslwr_s, **_wcslwr_s_l**	<string.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strlwr_s.cpp
// This program uses _strlwr_s and _strupr_s to create
// uppercase and lowercase copies of a mixed-case string.
//

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str[] = "The String to End All Strings!";
   char *copy1, *copy2;
   errno_t err;

   err = _strlwr_s( copy1 = _strdup(str), strlen(str) + 1);
   err = _strupr_s( copy2 = _strdup(str), strlen(str) + 1);

   printf( "Mixed: %s\n", str );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );

   free( copy1 );
   free( copy2 );

   return 0;
}
```

```
Mixed: The String to End All Strings!
Lower: the string to end all strings!
Upper: THE STRING TO END ALL STRINGS!
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l

3/1/2019 • 3 minutes to read • Edit Online

Appends characters of a string. More secure versions of these functions are available, see strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l .

> **IMPORTANT**
>
> **_mbsncat** and **_mbsncat_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strncat(
   char *strDest,
   const char *strSource,
   size_t count
);
wchar_t *wcsncat(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
unsigned char *_mbsncat(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count
);
unsigned char *_mbsncat_l(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
char *strncat(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
wchar_t *wcsncat(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncat(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncat_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest
Null-terminated destination string.

strSource
Null-terminated source string.

count
Number of characters to append.

locale
Locale to use.

Return Value

Returns a pointer to the destination string. No return value is reserved to indicate an error.

Remarks

The **strncat** function appends, at most, the first *count* characters of *strSource* to *strDest*. The initial character of *strSource* overwrites the terminating null character of *strDest*. If a null character appears in *strSource* before *count* characters are appended, **strncat** appends all characters from *strSource*, up to the null character. If *count* is greater than the length of *strSource*, the length of *strSource* is used in place of *count*. The all cases, the resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

> **IMPORTANT**
>
> **strncat** does not check for sufficient space in *strDest*; it is therefore a potential cause of buffer overruns. Keep in mind that *count* limits the number of characters appended; it is not a limit on the size of *strDest*. See the example below. For more information, see Avoiding Buffer Overruns.

wcsncat and **_mbsncat** are wide-character and multibyte-character versions of **strncat**. The string arguments and return value of **wcsncat** are wide-character strings; those of **_mbsncat** are multibyte-character strings. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In C++, these functions have template overloads. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncat	**strncat**	**_mbsnbcat**	**wcsncat**
_tcsncat_l	**_strncat_l**	**_mbsnbcat_l**	**_wcsncat_l**

> **NOTE**
>
> **_strncat_l** and **_wcsncat_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_tcsncat_l**.

Requirements

ROUTINE	REQUIRED HEADER
strncat	<string.h>
wcsncat	<string.h> or <wchar.h>
_mbsncat	<mbstring.h>
_mbsncat_l	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strncat.c
// Use strcat and strncat to append to a string.
#include <stdlib.h>

#define MAXSTRINGLEN 39

char string[MAXSTRINGLEN+1];
// or char *string = malloc(MAXSTRINGLEN+1);

void BadAppend( char suffix[], int n )
{
   strncat( string, suffix, n );
}

void GoodAppend( char suffix[], size_t n )
{
   strncat( string, suffix, __min( n, MAXSTRINGLEN-strlen(string)) );
}

int main( void )
{
   string[0] = '\0';
   printf( "string can hold up to %d characters\n", MAXSTRINGLEN );

   strcpy( string, "This is the initial string!" );
   // concatenate up to 20 characters...
   BadAppend( "Extra text to add to the string...", 20 );
   printf( "After BadAppend :  %s (%d chars)\n", string, strlen(string) );

   strcpy( string, "This is the initial string!" );
   // concatenate up to 20 characters...
   GoodAppend( "Extra text to add to the string...", 20 );
   printf( "After GoodAppend:  %s (%d chars)\n", string, strlen(string) );
}
```

Output

```
string can hold up to 39 characters
After BadAppend :  This is the initial string!Extra text to add to (47 chars)
After GoodAppend:  This is the initial string!Extra text t (39 chars)
```

Note that **BadAppend** caused a buffer overrun.

See also

String Manipulation
_mbsnbcat, _mbsnbcat_l
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l
Locale
Interpretation of Multibyte-Character Sequences

strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l

3/1/2019 • 6 minutes to read • Edit Online

Appends characters to a string. These versions of strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l have security enhancements, as described in Security Features in the CRT.

IMPORTANT

_mbsncat_s and **_mbsncat_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t strncat_s(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count
);
errno_t _strncat_s_l(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count,
   _locale_t locale
);
errno_t wcsncat_s(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count
);
errno_t _wcsncat_s_l(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
errno_t _mbsncat_s(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count
);
errno_t _mbsncat_s_l(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
errno_t strncat_s(
   char (&strDest)[size],
   const char *strSource,
```

```
    size_t count
); // C++ only
template <size_t size>
errno_t _strncat_s_l(
   char (&strDest)[size],
   const char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t wcsncat_s(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _wcsncat_s_l(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsncat_s(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsncat_s_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

strDest
Null-terminated destination string.

numberOfElements
Size of the destination buffer.

strSource
Null-terminated source string.

count
Number of characters to append, or _TRUNCATE.

locale
Locale to use.

Return Value

Returns 0 if successful, an error code on failure.

Error Conditions

STRDESTINATION	*NUMBEROFELEMENTS*	*STRSOURCE*	RETURN VALUE	CONTENTS OF *STRDESTINATION*
NULL or unterminated	any	any	**EINVAL**	not modified

STRDESTINATION	*NUMBEROFELEMENTS*	*STRSOURCE*	RETURN VALUE	CONTENTS OF *STRDESTINATION*
any	any	**NULL**	**EINVAL**	not modified
any	0, or too small	any	**ERANGE**	not modified

Remarks

These functions try to append the first *D* characters of *strSource* to the end of *strDest*, where *D* is the lesser of *count* and the length of *strSource*. If appending those *D* characters will fit within *strDest* (whose size is given as *numberOfElements*) and still leave room for a null terminator, then those characters are appended, starting at the original terminating null of *strDest*, and a new terminating null is appended; otherwise, *strDest*[0] is set to the null character and the invalid parameter handler is invoked, as described in Parameter Validation.

There is an exception to the above paragraph. If *count* is _TRUNCATE then as much of *strSource* as will fit is appended to *strDest* while still leaving room to append a terminating null.

For example,

```
char dst[5];
strncpy_s(dst, _countof(dst), "12", 2);
strncat_s(dst, _countof(dst), "34567", 3);
```

means that we are asking **strncat_s** to append three characters to two characters in a buffer five characters long; this would leave no space for the null terminator, hence **strncat_s** zeroes out the string and calls the invalid parameter handler.

If truncation behavior is needed, use **_TRUNCATE** or adjust the *size* parameter accordingly:

```
strncat_s(dst, _countof(dst), "34567", _TRUNCATE);
```

or

```
strncat_s(dst, _countof(dst), "34567", _countof(dst)-strlen(dst)-1);
```

In all cases, the resulting string is terminated with a null character. If copying takes place between strings that overlap, the behavior is undefined.

If *strSource* or *strDest* is **NULL**, or is *numberOfElements* is zero, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the function returns **EINVAL** without modifying its parameters.

wcsncat_s and **_mbsncat_s** are wide-character and multibyte-character versions of **strncat_s**. The string arguments and return value of **wcsncat_s** are wide-character strings; those of **_mbsncat_s** are multibyte-character strings. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-

secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncat_s	**strncat_s**	**_mbsnbcat_s**	**wcsncat_s**
_tcsncat_s_l	**_strncat_s_l**	**_mbsnbcat_s_l**	**_wcsncat_s_l**

_strncat_s_l and **_wcsncat_s_l** have no locale dependence; they are only provided for **_tcsncat_s_l**.

Requirements

ROUTINE	REQUIRED HEADER
strncat_s	<string.h>
wcsncat_s	<string.h> or <wchar.h>
_mbsncat_s, **_mbsncat_s_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strncat_s.cpp
// compile with: /MTd

// These #defines enable secure template overloads
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

errno_t strncat_s_tester( const char * initialDest,
                          const char * src,
                          int count )
{
   char dest[10];
   strcpy_s( dest, _countof(dest), initialDest );

   printf_s( "\n" );

   if ( count == _TRUNCATE )
      printf_s( "Appending '%s' to %d-byte buffer dest with truncation semantics\n",
                src, _countof(dest) );
   else
      printf_s( "Appending %d chars of '%s' to %d-byte buffer dest\n",
              count, src, _countof(dest) );
```

```
   printf_s( "    old contents of dest: '%s'\n", dest );

   errno_t err = strncat_s( dest, _countof(dest), src, count );

   printf_s( "    new contents of dest: '%s'\n", dest );

   return err;
}

void Examples()
{
   strncat_s_tester( "hi ", "there", 4 );
   strncat_s_tester( "hi ", "there", 5 );
   strncat_s_tester( "hi ", "there", 6 );

   printf_s( "\nDestination buffer too small:\n" );
   strncat_s_tester( "hello ", "there", 4 );

   printf_s( "\nTruncation examples:\n" );

   errno_t err = strncat_s_tester( "hello ", "there", _TRUNCATE );
   printf_s( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                           : "did not" );

   err = strncat_s_tester( "hello ", "!", _TRUNCATE );
   printf_s( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                           : "did not" );

   printf_s( "\nSecure template overload example:\n" );

   char dest[10] = "cats and ";
   strncat( dest, "dachshunds", 15 );
   // With secure template overloads enabled (see #define
   // at top of file), the preceding line is replaced by
   //    strncat_s( dest, _countof(dest), "dachshunds", 15 );
   // Instead of causing a buffer overrun, strncat_s invokes
   // the invalid parameter handler.
   // If secure template overloads were disabled, strncat would
   // append "dachshunds" and overrun the dest buffer.
   printf_s( "    new contents of dest: '%s'\n", dest );
}

void myInvalidParameterHandler(
   const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf_s(L"Invalid parameter handler invoked: %s\n", expression);
}

int main( void )
{
   _invalid_parameter_handler oldHandler, newHandler;

   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);
   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   Examples();
}
```

```
Appending 4 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi ther'

Appending 5 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi there'

Appending 6 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hi '
    new contents of dest: 'hi there'

Destination buffer too small:

Appending 4 chars of 'there' to 10-byte buffer dest
    old contents of dest: 'hello '
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''

Truncation examples:

Appending 'there' to 10-byte buffer dest with truncation semantics
    old contents of dest: 'hello '
    new contents of dest: 'hello the'
    truncation did occur

Appending '!' to 10-byte buffer dest with truncation semantics
    old contents of dest: 'hello '
    new contents of dest: 'hello !'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcat, _mbsnbcat_l
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

3/1/2019 • 3 minutes to read • Edit Online

Compares up to the specified count of characters of two strings.

> **IMPORTANT**
>
> **_mbsncmp** and **_mbsncmp_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int strncmp(
   const char *string1,
   const char *string2,
   size_t count
);
int wcsncmp(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsncmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _mbsncmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);int _mbsnbcmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
```

Parameters

string1, *string2*
Strings to compare.

count
Number of characters to compare.

locale
Locale to use.

Return Value

The return value indicates the relation of the substrings of *string1* and *string2* as follows.

RETURN VALUE	DESCRIPTION
< 0	*string1* substring less than *string2* substring
0	*string1* substring identical to *string2* substring
> 0	*string1* substring greater than *string2* substring

On a parameter validation error, **_mbsncmp** and **_mbsncmp_l** return **_NLSCMPERROR**, which is defined in <string.h> and <mbstring.h>.

Remarks

The **strncmp** function performs an ordinal comparison of at most the first *count* characters in *string1* and *string2* and returns a value indicating the relationship between the substrings. **strncmp** is a case-sensitive version of **_strnicmp**. **wcsncmp** and **_mbsncmp** are case-sensitive versions of **_wcsnicmp** and **_mbsnicmp**.

wcsncmp and **_mbsncmp** are wide-character and multibyte-character versions of **strncmp**. The arguments of **wcsncmp** are wide-character strings; those of **_mbsncmp** are multibyte-character strings. **_mbsncmp** recognizes multibyte-character sequences according to a multibyte code page and returns **_NLSCMPERROR** on an error.

Also, **_mbsncmp** and **_mbsncmp_l** validate parameters. If *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbsncmp** and **_mbsncmp_l** return **_NLSCMPERROR** and set **errno** to **EINVAL**. **strncmp** and **wcsncmp** do not validate their parameters. These functions behave identically otherwise.

The comparison behavior of **_mbsncmp** and **_mbsncmp_l** is affected by the setting of the **LC_CTYPE** category setting of the locale. This controls detection of leading and trailing bytes of multibyte characters. For more information, see setlocale. The **_mbsncmp** function uses the current locale for this locale-dependent behavior. The **_mbsncmp_l** function is identical except that it uses the *locale* parameter instead. For more information, see Locale. If the locale is a single-byte locale, the behavior of these functions is identical to **strncmp**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnccmp	**strncmp**	**_mbsncmp**	**wcsncmp**
_tcsncmp	**strncmp**	**_mbsnbcmp**	**wcsncmp**
_tccmp	Maps to macro or inline function	**_mbsncmp**	Maps to macro or inline function
not applicable	**not applicable**	**_mbsncmp_l**	**not applicable**

Requirements

ROUTINE	REQUIRED HEADER
strncmp	<string.h>

ROUTINE	REQUIRED HEADER
wcsncmp	<string.h> or <wchar.h>
_mbsncmp, **_mbsncmp_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strncmp.c
#include <string.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

int main( void )
{
   char tmp[20];
   int result;
   printf( "Compare strings:\n      %s\n      %s\n\n",
           string1, string2 );
   printf( "Function:   strncmp (first 10 characters only)\n" );
   result = strncmp( string1, string2 , 10 );
   if( result > 0 )
      strcpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, sizeof(tmp), "less than" );
   else
      strcpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:      String 1 is %s string 2\n\n", tmp );
   printf( "Function:   strnicmp _strnicmp (first 10 characters only)\n" );
   result = _strnicmp( string1, string2, 10 );
   if( result > 0 )
      strcpy_s( tmp, sizeof(tmp), "greater than" );
   else if( result < 0 )
      strcpy_s( tmp, sizeof(tmp), "less than" );
   else
      strcpy_s( tmp, sizeof(tmp), "equal to" );
   printf( "Result:      String 1 is %s string 2\n", tmp );
}
```

```
Compare strings:
      The quick brown dog jumps over the lazy fox
      The QUICK brown fox jumps over the lazy dog

Function:   strncmp (first 10 characters only)
Result:      String 1 is greater than string 2

Function:   strnicmp _strnicmp (first 10 characters only)
Result:      String 1 is equal to string 2
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcmp, _mbsnbcmp_l

_mbsnbicmp, _mbsnbicmp_l
strcmp, wcscmp, _mbscmp
strcoll Functions
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strncnt, _wcsncnt, _mbsnbcnt, _mbsnbcnt_l, _mbsnccnt, _mbsnccnt_l

11/9/2018 • 2 minutes to read • Edit Online

Returns the number of characters or bytes within a specified count.

> **IMPORTANT**
>
> **_mbsnbcnt**, **_mbsnbcnt_l**, **_mbsnccnt**, and **_mbsnccnt_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t _strncnt(
   const char *str,
   size_t count
);
size_t _wcsncnt(
   const wchar_t *str,
   size_t count
);
size_t _mbsnbcnt(
   const unsigned char *str,
   size_t count
);
size_t _mbsnbcnt_l(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
size_t _mbsnccnt(
   const unsigned char *str,
   size_t count
);
size_t _mbsnccnt_l(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
```

Parameters

str
String to be examined.

count
Number of characters or bytes to be examined in *str*.

locale
Locale to use.

Return Value

_mbsnbcnt and **_mbsnbcnt_l** return the number of bytes found in the first *count* of multibyte characters of *str*.

_mbsnccnt and **_mbsnccnt_l** return the number of characters found in the first *count* of bytes of *str*. If a null character is encountered before the examination of *str* has completed, they return the number of bytes or characters found before the null character. If *str* consists of fewer than *count* characters or bytes, they return the number of characters or bytes in the string. If *count* is less than zero, they return 0. In previous versions, these functions had a return value of type **int** rather than **size_t**.

_strncnt returns the number of characters in the first *count* bytes of the single-byte string *str*. **_wcsncnt** returns the number of characters in the first *count* wide characters of the wide-character string *str*.

Remarks

_mbsnbcnt and **_mbsnbcnt_l** count the number of bytes found in the first *count* of multibyte characters of *str*. **_mbsnbcnt** and **_mbsnbcnt_l** replace **mtob** and should be used in place of **mtob**.

_mbsnccnt and **_mbsnccnt_l** count the number of characters found in the first *count* of bytes of *str*. If **_mbsnccnt** and **_mbsnccnt_l** encounter a null character in the second byte of a double-byte character, the first byte is also considered to be null and is not included in the returned count value. **_mbsnccnt** and **_mbsnccnt_l** replace **btom** and should be used in place of **btom**.

If *str* is a **NULL** pointer or is *count* is 0, these functions invoke the invalid parameter handler as described in Parameter Validation, **errno** is set to **EINVAL**, and the function returns 0.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnbcnt	**_strncnt**	**_mbsnbcnt**	**_wcsncnt**
_tcsnccnt	**_strncnt**	**_mbsnbcnt**	n/a
_wcsncnt	n/a	n/a	**_mbsnbcnt**
_wcsncnt	n/a	n/a	**_mbsnccnt**
n/a	n/a	**_mbsnbcnt_l**	**_mbsnccnt_l**

Requirements

ROUTINE	REQUIRED HEADER
_mbsnbcnt	<mbstring.h>
_mbsnbcnt_l	<mbstring.h>
_mbsnccnt	<mbstring.h>
_mbsnccnt_l	<mbstring.h>

ROUTINE	REQUIRED HEADER
_strncnt	<tchar.h>
_wcsncnt	<tchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mbsnbcnt.c

#include  <mbstring.h>
#include  <stdio.h>

int main( void )
{
   unsigned char str[] = "This is a multibyte-character string.";
   unsigned int char_count, byte_count;
   char_count = _mbsnccnt( str, 10 );
   byte_count = _mbsnbcnt( str, 10 );
   if ( byte_count - char_count )
      printf( "The first 10 characters contain %d multibyte characters\n", char_count );
   else
      printf( "The first 10 characters are single-byte.\n");
}
```

Output

```
The first 10 characters are single-byte.
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcat, _mbsnbcat_l

_strncoll, _wcsncoll, _mbsncoll, _strncoll_l, _wcsncoll_l, _mbsncoll_l

10/31/2018 • 2 minutes to read • Edit Online

Compares strings by using locale-specific information.

> **IMPORTANT**
>
> **_mbsncoll** and **_mbsncoll_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _strncoll(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsncoll(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsncoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strncoll_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsncoll_l(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count,
   _locale_t locale
);
int _mbsncoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

count
The number of characters to compare.

locale

The locale to use.

Return Value

Each of these functions returns a value that indicates the relationship of the substrings of *string1* and *string2*, as follows.

RETURN VALUE	RELATIONSHIP OF STRING1 TO STRING2
< 0	*string1* is less than *string2*.
0	*string1* is identical to *string2*.
> 0	*string1* is greater than *string2*.

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either STRING.h or MBSTRING.h. **_wcsncoll** can fail if either *string1* or *string2* contains wide-character codes that are outside the domain of the collating sequence. When an error occurs, **_wcsncoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsncoll**, set **errno** to 0 and then check **errno** after you call **_wcsncoll**.

Remarks

Each of these functions performs a case-sensitive comparison of the first *count* characters in *string1* and *string2*, according to the code page that's currently in use. Use these functions only when there is a difference between the character set order and the lexicographic character order in the code page, and when this difference is of interest for the string comparison. The character set order is locale-dependent. The versions of these functions that don't have the **_l** suffix use the current locale, but the versions that have the **_l** suffix use the locale that's passed in. For more information, see Locale.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnccoll	**_strncoll**	**_mbsncoll**	**_wcsncoll**
_tcsncoll	**_strncoll**	_mbsnbcoll	**_wcsncoll**

Requirements

ROUTINE	REQUIRED HEADER
_strncoll, **_strncoll_l**	<string.h>
_wcsncoll, **_wcsncoll_l**	<wchar.h> or <string.h>
_mbsncoll, **_mbsncoll_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

See also

Locale
String Manipulation
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

3/1/2019 • 5 minutes to read • Edit Online

Copy characters of one string to another. More secure versions of these functions are available; see strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l.

> **IMPORTANT**
>
> **_mbsncpy** and **_mbsncpy_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strncpy(
   char *strDest,
   const char *strSource,
   size_t count
);
char *_strncpy_l(
   char *strDest,
   const char *strSource,
   size_t count,
   locale_t locale
);
wchar_t *wcsncpy(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
wchar_t *_wcsncpy_l(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count,
   locale_t locale
);
unsigned char *_mbsncpy(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count
);
unsigned char *_mbsncpy_l(
   unsigned char *strDest,
   const unsigned char *strSource,
   size_t count,
   _locale_t locale
);
template <size_t size>
char *strncpy(
   char (&strDest)[size],
   const char *strSource,
   size_t count
); // C++ only
template <size_t size>
char *_strncpy_l(
   char (&strDest)[size],
   const char *strSource,
```

```
    size_t count,
    locale_t locale
); // C++ only
template <size_t size>
wchar_t *wcsncpy(
    wchar_t (&strDest)[size],
    const wchar_t *strSource,
    size_t count
); // C++ only
template <size_t size>
wchar_t *_wcsncpy_l(
    wchar_t (&strDest)[size],
    const wchar_t *strSource,
    size_t count,
    locale_t locale
); // C++ only
template <size_t size>
unsigned char *_mbsncpy(
    unsigned char (&strDest)[size],
    const unsigned char *strSource,
    size_t count
); // C++ only
template <size_t size>
unsigned char *_mbsncpy_l(
    unsigned char (&strDest)[size],
    const unsigned char *strSource,
    size_t count,
    _locale_t locale
); // C++ only
```

Parameters

strDest
Destination string.

strSource
Source string.

count
Number of characters to be copied.

locale
Locale to use.

Return Value

Returns *strDest*. No return value is reserved to indicate an error.

Remarks

The **strncpy** function copies the initial *count* characters of *strSource* to *strDest* and returns *strDest*. If *count* is less than or equal to the length of *strSource*, a null character is not appended automatically to the copied string. If *count* is greater than the length of *strSource*, the destination string is padded with null characters up to length *count*. The behavior of **strncpy** is undefined if the source and destination strings overlap.

> **IMPORTANT**
> **strncpy** does not check for sufficient space in *strDest*; this makes it a potential cause of buffer overruns. The *count* argument limits the number of characters copied; it is not a limit on the size of *strDest*. See the following example. For more information, see Avoiding Buffer Overruns.

If *strDest* or *strSource* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

wcsncpy and **_mbsncpy** are wide-character and multibyte-character versions of **strncpy**. The arguments and return value of **wcsncpy** and **_mbsncpy** vary accordingly. These six functions behave identically otherwise.

The versions of these functions with the **_l** suffix are identical except that they use the locale passed in instead of the current locale for their locale-dependent behavior. For more information, see Locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncpy	**strncpy**	**_mbsnbcpy**	**wcsncpy**
_tcsncpy_l	**_strncpy_l**	**_mbsnbcpy_l**	**_wcsncpy_l**

> **NOTE**
> **_strncpy_l** and **_wcsncpy_l** have no locale dependence; they are provided just for **_tcsncpy_l** and are not intended to be called directly.

Requirements

ROUTINE	REQUIRED HEADER
strncpy	<string.h>
wcsncpy	<string.h> or <wchar.h>
_mbsncpy, **_mbsncpy_l**	<mbstring.h>

For additional platform compatibility information, see Compatibility.

Example

The following example demonstrates the use of **strncpy** and how it can be misused to cause program bugs and security issues. The compiler generates a warning for each call to **strncpy** similar to **crt_strncpy_x86.c(15) : warning C4996: 'strncpy': This function or variable may be unsafe. Consider using strncpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.**

```
// crt_strncpy_x86.c
// Use this command in an x86 developer command prompt to compile:
// cl /TC /W3 crt_strncpy_x86.c

#include <stdio.h>
#include <string.h>

int main() {
   char t[20];
   char s[20];
   char *p = 0, *q = 0;

   strcpy_s(s, sizeof(s), "AA BB CC");
   // Note: strncpy is deprecated; consider using strncpy_s instead
   strncpy(s, "aa", 2);     // "aa BB CC"         C4996
   strncpy(s + 3, "bb", 2); // "aa bb CC"         C4996
   strncpy(s, "ZZ", 3);     // "ZZ",              C4996
                            // count greater than strSource, null added
   printf("%s\n", s);

   strcpy_s(s, sizeof(s), "AA BB CC");
   p = strstr(s, "BB");
   q = strstr(s, "CC");
   strncpy(s, "aa", p - s - 1);   // "aa BB CC"   C4996
   strncpy(p, "bb", q - p - 1);   // "aa bb CC"   C4996
   strncpy(q, "cc",  q - s);      // "aa bb cc"   C4996
   strncpy(q, "dd", strlen(q));   // "aa bb dd"   C4996
   printf("%s\n", s);

   // some problems with strncpy
   strcpy_s(s, sizeof(s), "test");
   strncpy(t, "this is a very long string", 20 ); // C4996
   // Danger: at this point, t has no terminating null,
   // so the printf continues until it runs into one.
   // In this case, it will print "this is a very long test"
   printf("%s\n", t);

   strcpy_s(t, sizeof(t), "dogs like cats");
   printf("%s\n", t);

   strncpy(t + 10, "to chase cars.", 14); // C4996
   printf("%s\n", t);

   // strncpy has caused a buffer overrun and corrupted string s
   printf("Buffer overrun: s = '%s' (should be 'test')\n", s);
   // Since the stack grows from higher to lower addresses, buffer
   // overruns can corrupt function return addresses on the stack,
   // which can be exploited to run arbitrary code.
}
```

Output

```
ZZ
aa bb dd
this is a very long test
dogs like cats
dogs like to chase cars.
Buffer overrun: s = 'ars.' (should be 'test')
```

The layout of automatic variables and the level of error detection and code protection can vary with changed compiler settings. This example may have different results when built in other compilation environments or with other compiler options.

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcpy, _mbsnbcpy_l
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strcpy_s, wcscpy_s, _mbscpy_s

strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

3/1/2019 • 6 minutes to read • Edit Online

Copies characters of one string to another. These versions of strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l have security enhancements, as described in Security Features in the CRT.

IMPORTANT

_mbsncpy_s and **_mbsncpy_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t strncpy_s(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count
);
errno_t _strncpy_s_l(
   char *strDest,
   size_t numberOfElements,
   const char *strSource,
   size_t count,
   _locale_t locale
);
errno_t wcsncpy_s(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count
);
errno_t _wcsncpy_s_l(
   wchar_t *strDest,
   size_t numberOfElements,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
errno_t _mbsncpy_s(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count
);
errno_t _mbsncpy_s_l(
   unsigned char *strDest,
   size_t numberOfElements,
   const unsigned char *strSource,
   size_t count,
   locale_t locale
);
template <size_t size>
errno_t strncpy_s(
   char (&strDest)[size],
   const char *strSource,
```

```
    size_t count
); // C++ only
template <size_t size>
errno_t _strncpy_s_l(
   char (&strDest)[size],
   const char *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t wcsncpy_s(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _wcsncpy_s_l(
   wchar_t (&strDest)[size],
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsncpy_s(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count
); // C++ only
template <size_t size>
errno_t _mbsncpy_s_l(
   unsigned char (&strDest)[size],
   const unsigned char *strSource,
   size_t count,
   locale_t locale
); // C++ only
```

Parameters

strDest
Destination string.

numberOfElements
The size of the destination string, in characters.

strSource
Source string.

count
Number of characters to be copied, or _TRUNCATE.

locale
The locale to use.

Return Value

Zero if successful, **STRUNCATE** if truncation occurred, otherwise an error code.

Error Conditions

STRDEST	*NUMBEROFELEMENTS*	*STRSOURCE*	RETURN VALUE	CONTENTS OF *STRDEST*
NULL	any	any	**EINVAL**	not modified

STRDEST	*NUMBEROFELEMENTS*	*STRSOURCE*	RETURN VALUE	CONTENTS OF *STRDEST*
any	any	**NULL**	**EINVAL**	*strDest*[0] set to 0
any	0	any	**EINVAL**	not modified
not **NULL**	too small	any	**ERANGE**	*strDest*[0] set to 0

Remarks

These functions try to copy the first *D* characters of *strSource* to *strDest*, where *D* is the lesser of *count* and the length of *strSource*. If those *D* characters will fit within *strDest* (whose size is given as *numberOfElements*) and still leave room for a null terminator, then those characters are copied and a terminating null is appended; otherwise, *strDest*[0] is set to the null character and the invalid parameter handler is invoked, as described in Parameter Validation.

There is an exception to the above paragraph. If *count* is **_TRUNCATE**, then as much of *strSource* as will fit into *strDest* is copied while still leaving room for the terminating null which is always appended.

For example,

```
char dst[5];
strncpy_s(dst, 5, "a long string", 5);
```

means that we are asking **strncpy_s** to copy five characters into a buffer five bytes long; this would leave no space for the null terminator, hence **strncpy_s** zeroes out the string and calls the invalid parameter handler.

If truncation behavior is needed, use **_TRUNCATE** or (*size* - 1):

```
strncpy_s(dst, 5, "a long string", _TRUNCATE);
strncpy_s(dst, 5, "a long string", 4);
```

Note that unlike **strncpy**, if *count* is greater than the length of *strSource*, the destination string is NOT padded with null characters up to length *count*.

The behavior of **strncpy_s** is undefined if the source and destination strings overlap.

If *strDest* or *strSource* is **NULL**, or *numberOfElements* is 0, the invalid parameter handler is invoked. If execution is allowed to continue, the function returns **EINVAL** and sets **errno** to **EINVAL**.

wcsncpy_s and **_mbsncpy_s** are wide-character and multibyte-character versions of **strncpy_s**. The arguments and return value of **wcsncpy_s** and **mbsncpy_s** do vary accordingly. These six functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use

_CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncpy_s	**strncpy_s**	**_mbsnbcpy_s**	**wcsncpy_s**
_tcsncpy_s_l	**_strncpy_s_l**	**_mbsnbcpy_s_l**	**_wcsncpy_s_l**

> **NOTE**
>
> **_strncpy_s_l**, **_wcsncpy_s_l** and **_mbsncpy_s_l** have no locale dependence and are provided just for **_tcsncpy_s_l** and are not intended to be called directly.

Requirements

ROUTINE	REQUIRED HEADER
strncpy_s, **_strncpy_s_l**	<string.h>
wcsncpy_s, **_wcsncpy_s_l**	<string.h> or <wchar.h>
_mbsncpy_s, **_mbsncpy_s_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strncpy_s_1.cpp
// compile with: /MTd

// these #defines enable secure template overloads
// (see last part of Examples() below)
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <crtdbg.h>  // For _CrtSetReportMode
#include <errno.h>

// This example uses a 10-byte destination buffer.

errno_t strncpy_s_tester( const char * src,
                          int count )
{
   char dest[10];

   printf( "\n" );

   if ( count == _TRUNCATE )
      printf( "Copying '%s' to %d-byte buffer dest with truncation semantics\n",
               src, _countof(dest) );
   else
      printf( "Copying %d chars of '%s' to %d-byte buffer dest\n",
              count, src, _countof(dest) );
```

```
   errno_t err = strncpy_s( dest, _countof(dest), src, count );

   printf( "    new contents of dest: '%s'\n", dest );

   return err;
}

void Examples()
{
   strncpy_s_tester( "howdy", 4 );
   strncpy_s_tester( "howdy", 5 );
   strncpy_s_tester( "howdy", 6 );

   printf( "\nDestination buffer too small:\n" );
   strncpy_s_tester( "Hi there!!", 10 );

   printf( "\nTruncation examples:\n" );

   errno_t err = strncpy_s_tester( "How do you do?", _TRUNCATE );
   printf( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                         : "did not" );

   err = strncpy_s_tester( "Howdy.", _TRUNCATE );
   printf( "    truncation %s occur\n", err == STRUNCATE ? "did"
                                                         : "did not" );

   printf( "\nSecure template overload example:\n" );

   char dest[10];
   strncpy( dest, "very very very long", 15 );
   // With secure template overloads enabled (see #defines at
   // top of file), the preceding line is replaced by
   //    strncpy_s( dest, _countof(dest), "very very very long", 15 );
   // Instead of causing a buffer overrun, strncpy_s invokes
   // the invalid parameter handler.
   // If secure template overloads were disabled, strncpy would
   // copy 15 characters and overrun the dest buffer.
   printf( "    new contents of dest: '%s'\n", dest );
}

void myInvalidParameterHandler(
   const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf(L"Invalid parameter handler invoked: %s\n", expression);
}

int main( void )
{
   _invalid_parameter_handler oldHandler, newHandler;

   newHandler = myInvalidParameterHandler;
   oldHandler = _set_invalid_parameter_handler(newHandler);
   // Disable the message box for assertions.
   _CrtSetReportMode(_CRT_ASSERT, 0);

   Examples();
}
```

```
Copying 4 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howd'

Copying 5 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howdy'

Copying 6 chars of 'howdy' to 10-byte buffer dest
    new contents of dest: 'howdy'

Destination buffer too small:

Copying 10 chars of 'Hi there!!' to 10-byte buffer dest
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''

Truncation examples:

Copying 'How do you do?' to 10-byte buffer dest with truncation semantics
    new contents of dest: 'How do yo'
    truncation did occur

Copying 'Howdy.' to 10-byte buffer dest with truncation semantics
    new contents of dest: 'Howdy.'
    truncation did not occur

Secure template overload example:
Invalid parameter handler invoked: (L"Buffer is too small" && 0)
    new contents of dest: ''
```

Example

```
// crt_strncpy_s_2.c
// contrasts strncpy and strncpy_s

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char a[20] = "test";
   char s[20];

   // simple strncpy usage:

   strcpy_s( s, 20, "dogs like cats" );
   printf( "Original string:\n   '%s'\n", s );

   // Here we can't use strncpy_s since we don't
   // want null termination
   strncpy( s, "mice", 4 );
   printf( "After strncpy (no null-termination):\n   '%s'\n", s );
   strncpy( s+5, "love", 4 );
   printf( "After strncpy into middle of string:\n   '%s'\n", s );

   // If we use strncpy_s, the string is terminated
   strncpy_s( s, _countof(s), "mice", 4 );
   printf( "After strncpy_s (with null-termination):\n   '%s'\n", s );

}
```

```
Original string:
   'dogs like cats'
After strncpy (no null-termination):
   'mice like cats'
After strncpy into middle of string:
   'mice love cats'
After strncpy_s (with null-termination):
   'mice'
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbcpy, _mbsnbcpy_l
strcat_s, wcscat_s, _mbscat_s
strcmp, wcscmp, _mbscmp
strcpy_s, wcscpy_s, _mbscpy_s
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l

11/9/2018 • 2 minutes to read • Edit Online

Finds the next character in a string.

> **IMPORTANT**
> **_mbsnextc** and **_mbsnextc_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
unsigned int _strnextc(
   const char *str
);
unsigned int _wscnextc(
   const wchar_t *str
);
unsigned int _mbsnextc(
   const unsigned char *str
);
unsigned int _mbsnextc_l(
   const unsigned char *str,
   _locale_t locale
);
```

Parameters

str
Source string.

locale
Locale to use.

Return Value

Each of these functions returns the integer value of the next character in *str*.

Remarks

The **_mbsnextc** function returns the integer value of the next multibyte character in *str*, without advancing the string pointer. **_mbsnextc** recognizes multibyte-character sequences according to the multibyte code page currently in use.

If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns 0.

Security Note This API incurs a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnextc	**_strnextc**	**_mbsnextc**	**_wcsnextc**

_strnextc and **_wcsnextc** are single-byte-character string and wide-character string versions of **_mbsnextc**. **_wcsnextc** returns the integer value of the next wide character in *str*; **_strnextc** returns the integer value of the next single-byte character in *str*. **_strnextc** and **_wcsnextc** are provided only for this mapping and should not be used otherwise. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

_mbsnextc_l is identical except that it uses the locale parameter passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_mbsnextc	<mbstring.h>
_mbsnextc_l	<mbstring.h>
_strnextc	<tchar.h>
_wcsnextc	<tchar.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_strdec, _wcsdec, _mbsdec, _mbsdec_l
_strinc, _wcsinc, _mbsinc, _mbsinc_l
_strninc, _wcsninc, _mbsninc, _mbsninc_l

strnicmp, wcsnicmp

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l instead.

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

10/31/2018 • 2 minutes to read • Edit Online

Compares the specified number of characters of two strings without regard to case.

IMPORTANT

_mbsnicmp and **_mbsnicmp_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _strnicmp(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsnicmp(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count
);
int _mbsnicmp(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strnicmp_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsnicmp_l(
   const wchar_t *string1,
   const wchar_t *string2,
   size_t count,
   _locale_t locale
);
int _mbsnicmp_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare.

count
Number of characters to compare.

locale

Locale to use.

Return Value

Indicates the relationship between the substrings, as follows.

RETURN VALUE	DESCRIPTION
< 0	*string1* substring is less than *string2* substring.
0	*string1* substring is identical to *string2* substring.
> 0	*string1* substring is greater than *string2* substring.

On a parameter validation error, these functions return **_NLSCMPERROR**, which is defined in <string.h> and <mbstring.h>.

Remarks

The **_strnicmp** function ordinally compares, at most, the first *count* characters of *string1* and *string2*. The comparison is performed without regard to case by converting each character to lowercase. **_strnicmp** is a case-insensitive version of **strncmp**. The comparison ends if a terminating null character is reached in either string before *count* characters are compared. If the strings are equal when a terminating null character is reached in either string before *count* characters are compared, the shorter string is lesser.

The characters from 91 to 96 in the ASCII table ('[', '\', ']', '^', '_', and '`') evaluate as less than any alphabetic character. This ordering is identical to that of **stricmp**.

_wcsnicmp and **_mbsnicmp** are wide-character and multibyte-character versions of **_strnicmp**. The arguments of **_wcsnicmp** are wide-character strings; those of **_mbsnicmp** are multibyte-character strings. **_mbsnicmp** recognizes multibyte-character sequences according to the current multibyte code page and returns **_NLSCMPERROR** on an error. For more information, see Code Pages. These three functions behave identically otherwise. These functions are affected by the locale setting—the versions that don't have the **_l** suffix use the current locale for their locale-dependent behavior; the versions that do have the **_l** suffix instead use the *locale* that's passed in. For more information, see Locale.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncicmp	**_strnicmp**	**_mbsnicmp**	**_wcsnicmp**
_tcsnicmp	**_strnicmp**	**_mbsnbicmp**	**_wcsnicmp**
_tcsncicmp_l	**_strnicmp_l**	**_mbsnicmp_l**	**_wcsnicmp_l**

Requirements

ROUTINE	REQUIRED HEADER
_strnicmp, **_strnicmp_l**	<string.h>
_wcsnicmp, **_wcsnicmp_l**	<string.h> or <wchar.h>
_mbsnicmp, **_mbsnicmp_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

See the example for strncmp.

See also

String Manipulation
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

_strnicoll, _wcsnicoll, _mbsnicoll, _strnicoll_l, _wcsnicoll_l, _mbsnicoll_l

10/31/2018 • 2 minutes to read • Edit Online

Compares strings by using locale-specific information.

IMPORTANT

_mbsnicoll and **_mbsnicoll_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _strnicoll(
   const char *string1,
   const char *string2,
   size_t count
);
int _wcsnicoll(
   const wchar_t *string1,
   const wchar_t *string2 ,
   size_t count
);
int _mbsnicoll(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count
);
int _strnicoll_l(
   const char *string1,
   const char *string2,
   size_t count,
   _locale_t locale
);
int _wcsnicoll_l(
   const wchar_t *string1,
   const wchar_t *string2 ,
   size_t count,
   _locale_t locale
);
int _mbsnicoll_l(
   const unsigned char *string1,
   const unsigned char *string2,
   size_t count,
   _locale_t locale
);
```

Parameters

string1, *string2*
Null-terminated strings to compare

count
Number of characters to compare

locale

The locale to use.

Return Value

Each of these functions returns a value indicating the relationship of the substrings of *string1* and *string2*, as follows.

RETURN VALUE	RELATIONSHIP OF STRING1 TO STRING2
< 0	*string1* less than *string2*
0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

Each of these functions returns **_NLSCMPERROR**. To use **_NLSCMPERROR**, include either STRING.H or MBSTRING.H. **_wcsnicoll** can fail if either *string1* or *string2* contains wide-character codes outside the domain of the collating sequence. When an error occurs, **_wcsnicoll** may set **errno** to **EINVAL**. To check for an error on a call to **_wcsnicoll**, set **errno** to 0 and then check **errno** after calling **_wcsnicoll**.

Remarks

Each of these functions performs a case-insensitive comparison of the first *count* characters in *string1* and *string2* according to the code page. These functions should be used only when there is a difference between the character set order and the lexicographic character order in the code page and this difference is of interest for the string comparison. The versions of these functions without the **_l** suffix use the current locale and code page. The versions with the **_l** suffix are identical except that they use the locale passed in instead. For more information, see Locale.

All of these functions validate their parameters. If either *string1* or *string2* is a null pointer, or if count is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return **_NLSCMPERROR** and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsncicoll	**_strnicoll**	**_mbsnbicoll**	**_wcsnicoll**
_tcsnicoll	**_strnicoll**	_mbsnbicoll	**_wcsnicoll**
_tcsnicoll_l	**_strnicoll_l**	**_mbsnbicoll_l**	**_wcsnicoll_l**

Requirements

ROUTINE	REQUIRED HEADER
_strnicoll, **_strnicoll_l**	<string.h>
_wcsnicoll, **_wcsnicoll_l**	<wchar.h> or <string.h>
_mbsnicoll, **_mbsnicoll_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

See also

Locale
String Manipulation
strcoll Functions
localeconv
_mbsnbcoll, _mbsnbcoll_l, _mbsnbicoll, _mbsnbicoll_l
setlocale, _wsetlocale
strcmp, wcscmp, _mbscmp
_stricmp, _wcsicmp, _mbsicmp, _stricmp_l, _wcsicmp_l, _mbsicmp_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

_strninc, _wcsninc, _mbsninc, _mbsninc_l

10/31/2018 • 2 minutes to read • Edit Online

Advances a string pointer by **n** characters.

> **IMPORTANT**
>
> **_mbsninc** and **_mbsninc_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strninc(
   const char *str,
   size_t count
);
wchar_t *_wcsninc(
   const wchar_t *str,
   size_t count
);
unsigned char *_mbsninc(
   const unsigned char *str,
   size_t count
);
unsigned char *_mbsninc(
   const unsigned char *str,
   size_t count,
   _locale_t locale
);
```

Parameters

str
Source string.

count
Number of characters to increment a string pointer.

locale
Locale to use.

Return Value

Each of these routines returns a pointer to *str* after *str* has been incremented by *count* characters or **NULL** if the supplied pointer is **NULL**. If *count* is greater than or equal to the number of characters in *str*, the result is undefined.

Remarks

The **_mbsninc** function increments *str* by *count* multibyte characters. **_mbsninc** recognizes multibyte-character sequences according to the multibyte code page currently in use.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsninc	**_strninc**	**_mbsninc**	**_wcsninc**

_strninc and **_wcsninc** are single-byte-character string and wide-character string versions of **_mbsninc**. **_wcsninc** and **_strninc** are provided only for this mapping and should not be used otherwise. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

_mbsninc_l is identical except that it uses the locale parameter passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_mbsninc	<mbstring.h>
_mbsninc_l	<mbstring.h>
_strninc	<tchar.h>
_wcsninc	<tchar.h>

For more compatibility information, see Compatibility.

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_strdec, _wcsdec, _mbsdec, _mbsdec_l
_strinc, _wcsinc, _mbsinc, _mbsinc_l
_strnextc, _wcsnextc, _mbsnextc, _mbsnextc_l

strnlen, strnlen_s, wcsnlen, wcsnlen_s, _mbsnlen, _mbsnlen_l, _mbstrnlen, _mbstrnlen_l

3/1/2019 • 4 minutes to read • Edit Online

Gets the length of a string by using the current locale or one that has been passed in. These are more secure versions of strlen, wcslen, _mbslen, _mbslen_l, _mbstrlen, _mbstrlen_l.

> **IMPORTANT**
>
> **_mbsnlen**, **_mbsnlen_l**, **_mbstrnlen**, and **_mbstrnlen_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t strnlen(
   const char *str,
   size_t numberOfElements
);
size_t strnlen_s(
   const char *str,
   size_t numberOfElements
);
size_t wcsnlen(
   const wchar_t *str,
   size_t numberOfElements
);
size_t wcsnlen_s(
   const wchar_t *str,
   size_t numberOfElements
);
size_t _mbsnlen(
   const unsigned char *str,
   size_t numberOfElements
);
size_t _mbsnlen_l(
   const unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
size_t _mbstrnlen(
   const char *str,
   size_t numberOfElements
);
size_t _mbstrnlen_l(
   const char *str,
   size_t numberOfElements,
   _locale_t locale
);
```

Parameters

str
Null-terminated string.

numberOfElements
The size of the string buffer.

locale
Locale to use.

Return Value

These functions return the number of characters in the string, not including the terminating null character. If there is no null terminator within the first *numberOfElements* bytes of the string (or wide characters for **wcsnlen**), then *numberOfElements* is returned to indicate the error condition; null-terminated strings have lengths that are strictly less than *numberOfElements*.

_mbstrnlen and **_mbstrnlen_l** return -1 if the string contains an invalid multibyte character.

Remarks

> **NOTE**
> **strnlen** is not a replacement for **strlen**; **strnlen** is intended to be used only to calculate the size of incoming untrusted data in a buffer of known size—for example, a network packet. **strnlen** calculates the length but doesn't walk past the end of the buffer if the string is unterminated. For other situations, use **strlen**. (The same applies to **wcsnlen**, **_mbsnlen**, and **_mbstrnlen**.)

Each of these functions returns the number of characters in *str*, not including the terminating null character. However, **strnlen** and **strnlen_s** interpret the string as a single-byte character string and therefore, the return value is always equal to the number of bytes, even if the string contains multibyte characters. **wcsnlen** and **wcsnlen_s** are wide-character versions of **strnlen** and **strnlen_s** respectively; the arguments for **wcsnlen** and **wcsnlen_s** are wide-character strings and the count of characters are in wide-character units. Otherwise, **wcsnlen** and **strnlen** behave identically, as do **strnlen_s** and **wcsnlen_s**.

strnlen, **wcsnlen**, and **_mbsnlen** do not validate their parameters. If *str* is **NULL**, an access violation occurs.

strnlen_s and **wcsnlen_s** validate their parameters. If *str* is **NULL**, the functions return 0.

_mbstrnlen also validates its parameters. If *str* is **NULL**, or if *numberOfElements* is greater than **INT_MAX**, **_mbstrnlen** generates an invalid parameter exception, as described in Parameter Validation. If execution is allowed to continue, **_mbstrnlen** sets **errno** to **EINVAL** and returns -1.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnlen	**strnlen**	**strnlen**	**wcsnlen**
_tcscnlen	**strnlen**	**_mbsnlen**	**wcsnlen**
_tcscnlen_l	**strnlen**	**_mbsnlen_l**	**wcsnlen**

_mbsnlen and **_mbstrnlen** return the number of multibyte characters in a multibyte-character string. **_mbsnlen** recognizes multibyte-character sequences according to the multibyte code page that's currently in use or according to the locale that's passed in; it does not test for multibyte-character validity. **_mbstrnlen** tests for multibyte-character validity and recognizes multibyte-character sequences. If the string that's passed to **_mbstrnlen** contains an invalid multibyte character, **errno** is set to **EILSEQ**.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions are identical, except that the ones that don't

have the **_l** suffix use the current locale for this locale-dependent behavior and the versions that have the **_l** suffix instead use the locale parameter that's passed in. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
strnlen, **strnlen_s**	<string.h>
wcsnlen, **wcsnlen_s**	<string.h> or <wchar.h>
_mbsnlen, **_mbsnlen_l**	<mbstring.h>
_mbstrnlen, **_mbstrnlen_l**	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strnlen.c

#include <string.h>

int main()
{
   // str1 is 82 characters long. str2 is 159 characters long

   char* str1 = "The length of a string is the number of characters\n"
                "excluding the terminating null.";
   char* str2 = "strnlen takes a maximum size. If the string is longer\n"
                "than the maximum size specified, the maximum size is\n"
                "returned rather than the actual size of the string.";
   size_t len;
   size_t maxsize = 100;

   len = strnlen(str1, maxsize);
   printf("%s\n Length: %d \n\n", str1, len);

   len = strnlen(str2, maxsize);
   printf("%s\n Length: %d \n", str2, len);
}
```

```
The length of a string is the number of characters
excluding the terminating null.
Length: 82

strnlen takes a maximum size. If the string is longer
than the maximum size specified, the maximum size is
returned rather than the actual size of the string.
Length: 100
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
setlocale, _wsetlocale

strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strcoll Functions
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strnset, wcsnset

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l or security-enhanced _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l instead.

_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

3/1/2019 • 2 minutes to read • Edit Online

Initializes characters of a string to a given character. More secure versions of these functions exist; see _strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l.

> **IMPORTANT**
>
> **_mbsnset** and **_mbsnset_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strnset(
   char *str,
   int c,
   size_t count
);
char *_strnset_l(
   char *str,
   int c,
   size_t count,
   locale_t locale
);
wchar_t *_wcsnset(
   wchar_t *str,
   wchar_t c,
   size_t count
);
wchar_t *_wcsnset_l(
   wchar_t *str,
   wchar_t c,
   size_t count,
   _locale_t locale
);
unsigned char *_mbsnset(
   unsigned char *str,
   unsigned int c,
   size_t count
);
unsigned char *_mbsnset_l(
   unsigned char *str,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str

String to be altered.

c

Character setting.

count
Number of characters to be set.

locale
Locale to use.

Return Value

Returns a pointer to the altered string.

Remarks

The **_strnset** function sets, at most, the first *count* characters of *str* to *c* (converted to **char**). If *count* is greater than the length of *str*, the length of *str* is used instead of *count*.

_wcsnset and **_mbsnset** are wide-character and multibyte-character versions of **_strnset**. The string arguments and return value of **_wcsnset** are wide-character strings; those of **_mbsnset** are multibyte-character strings. These three functions behave identically otherwise.

_mbsnset validates its parameters; if *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, **_mbsnset** returns **NULL** and sets **errno** to **EINVAL**. **_strnset** and **_wcsnset** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnset	**_strnset**	**_mbsnbset**	**_wcsnset**
_tcsnset_l	**_strnset_l**	**_mbsnbset_l**	**_wcsnset_l**

Requirements

ROUTINE	REQUIRED HEADER
_strnset	<string.h>
_strnset_l	<tchar.h>
_wcsnset	<string.h> or <wchar.h>
_wcsnset_l	<tchar.h>
_mbsnset, **_mbsnset_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strnset.c
// compile with: /W3
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   _strnset( string, '*', 4 ); // C4996
   // Note: _strnset is deprecated; consider using _strnset_s
   printf( "After:  %s\n", string );
}
```

```
Before: This is a test
After:  **** is a test
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

_strnset_s, _strnset_s_l, _wcsnset_s, _wcsnset_s_l, _mbsnset_s, _mbsnset_s_l

3/1/2019 • 2 minutes to read • Edit Online

Initializes characters of a string to a given character. These versions of _strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbsnset_s** and **_mbsnset_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _strnset_s(
   char *str,
   size_t numberOfElements,
   int c,
   size_t count
);
errno_t _strnset_s_l(
   char *str,
   size_t numberOfElements,
   int c,
   size_t count,
   locale_t locale
);
errno_t _wcsnset_s(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   size_t count
);
errno_t _wcsnset_s_l(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   size_t count,
   _locale_t locale
);
errno_t _mbsnset_s(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   size_t count
);
errno_t _mbsnset_s_l(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   size_t count,
   _locale_t locale
);
```

Parameters

str
String to be altered.

numberOfElements
The size of the *str* buffer.

c
Character setting.

count
Number of characters to be set.

locale
Locale to use.

Return Value

Zero if successful, otherwise an error code.

These functions validate their arguments. If *str* is not a valid null-terminated string or the size argument is less than or equal to 0, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return an error code and set **errno** to that error code. The default error code is **EINVAL** if a more specific value does not apply.

Remarks

These functions set, at most, the first *count* characters of *str* to *c*. If *count* is greater than the size of *str*, the size of *str* is used instead of *count*. An error occurs if *count* is greater than *numberOfElements* and both those parameters are greater than the size of *str*.

_wcsnset_s and **_mbsnset_s** are wide-character and multibyte-character versions of **_strnset_s**. The string argument of **_wcsnset_s** is a wide-character string; that of **_mbsnset_s** is amultibyte-character string. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsnset_s	**_strnset_s**	**_mbsnbset_s**	**_wcsnset_s**
_tcsnset_s_l	**_strnset_s_l**	**_mbsnbset_s_l**	**_wcsnset_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_strnset_s	<string.h>

ROUTINE	REQUIRED HEADER
_strnset_s_l	<tchar.h>
_wcsnset_s	<string.h> or <wchar.h>
_wcsnset_s_l	<tchar.h>
_mbsnset_s, **_mbsnset_s_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strnset_s.c
#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   _strnset_s( string, sizeof(string), '*', 4 );
   printf( "After:  %s\n", string );
}
```

```
Before: This is a test
After:  **** is a test
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l

10/31/2018 • 3 minutes to read • Edit Online

Scans strings for characters in specified character sets.

IMPORTANT

`_mbspbrk` and `_mbspbrk_l` cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strpbrk(
   const char *str,
   const char *strCharSet
); // C only
char *strpbrk(
   char *str,
   const char *strCharSet
); // C++ only
const char *strpbrk(
   const char *str,
   const char *strCharSet
); // C++ only
wchar_t *wcspbrk(
   const wchar_t *str,
   const wchar_t *strCharSet
); // C only
wchar_t *wcspbrk(
   wchar_t *str,
   const wchar_t *strCharSet
); // C++ only
const wchar_t *wcspbrk(
   const wchar_t *str,
   const wchar_t *strCharSet
); // C++ only
unsigned char *_mbspbrk(
   const unsigned char *str,
   const unsigned char *strCharSet
); // C only
unsigned char *_mbspbrk(
   unsigned char *str,
   const unsigned char *strCharSet
); // C++ only
const unsigned char *_mbspbrk(
   const unsigned char *str,
   const unsigned char *strCharSet
); // C++ only
unsigned char *_mbspbrk_l(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
); // C only
unsigned char *_mbspbrk_l(
   unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
); // C++ only
const unsigned char *_mbspbrk_l(
   const unsigned char *str,
   const unsigned char* strCharSet,
   _locale_t locale
); // C++ only
```

Parameters

str

Null-terminated, searched string.

strCharSet

Null-terminated character set.

locale

Locale to use.

Return Value

Returns a pointer to the first occurrence of any character from *strCharSet* in *str*, or a NULL pointer if the two string arguments have no characters in common.

Remarks

The `strpbrk` function returns a pointer to the first occurrence of a character in *str* that belongs to the set of characters in *strCharSet*. The search does not include the terminating null character.

`wcspbrk` and `_mbspbrk` are wide-character and multibyte-character versions of `strpbrk`. The arguments and return value of `wcspbrk` are wide-character strings; those of `_mbspbrk` are multibyte-character strings.

`_mbspbrk` validates its parameters. If *str* or *strCharSet* is NULL, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, `_mbspbrk` returns NULL and sets `errno` to EINVAL. `strpbrk` and `wcspbrk` do not validate their parameters. These three functions behave identically otherwise.

`_mbspbrk` is similar to `_mbscspn` except that `_mbspbrk` returns a pointer rather than a value of type size_t.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CRT_CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

The output value is affected by the setting of the LC_CTYPE category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the version with the **_l** suffix is identical except that it uses the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tcspbrk`	`strpbrk`	`_mbspbrk`	`wcspbrk`
n/a	**n/a**	`_mbspbrk_l`	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
`strpbrk`	<string.h>
`wcspbrk`	<string.h> or <wchar.h>
`_mbspbrk`, `_mbspbrk_l`	<mbstring.h>

For more information about compatibility, see Compatibility.

Example

```
// crt_strpbrk.c

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
   char *result = NULL;

   // Return pointer to first digit in "string".
   printf( "1: %s\n", string );
   result = strpbrk( string, "0123456789" );
   printf( "2: %s\n", result++ );
   result = strpbrk( result, "0123456789" );
   printf( "3: %s\n", result++ );
   result = strpbrk( result, "0123456789" );
   printf( "4: %s\n", result );
}
```

```
1: The 3 men and 2 boys ate 5 pigs

2: 3 men and 2 boys ate 5 pigs

3: 2 boys ate 5 pigs

4: 5 pigs
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strchr, wcschr, _mbschr, _mbschr_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

3/1/2019 • 2 minutes to read • Edit Online

Scans a string for the last occurrence of a character.

IMPORTANT

`_mbsrchr` and `_mbsrchr_l` cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strrchr(
   const char *str,
   int c
); // C only
char *strrchr(
   char *str,
   int c
); // C++ only
const char *strrchr(
   const char *str,
   int c
); // C++ only
wchar_t *wcsrchr(
   const wchar_t *str,
   wchar_t c
); // C only
wchar_t *wcsrchr(
   wchar_t *str,
   wchar_t c
); // C++ only
const wchar_t *wcsrchr(
   const wchar_t *str,
   wchar_t c
); // C++ only
unsigned char *_mbsrchr(
   const unsigned char *str,
   unsigned int c
); // C only
unsigned char *_mbsrchr(
   unsigned char *str,
   unsigned int c
); // C++ only
const unsigned char *_mbsrchr(
   const unsigned char *str,
   unsigned int c
); // C++ only
unsigned char *_mbsrchr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C only
unsigned char *_mbsrchr_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
const unsigned char *_mbsrchr_l(
   const unsigned char *str,
   unsigned int c,
   _locale_t locale
); // C++ only
```

Parameters

str
Null-terminated string to search.

c
Character to be located.

locale
Locale to use.

Return Value

Returns a pointer to the last occurrence of *c* in *str*, or NULL if *c* is not found.

Remarks

The `strrchr` function finds the last occurrence of *c* (converted to **char**) in *str*. The search includes the terminating null character.

`wcsrchr` and `_mbsrchr` are wide-character and multibyte-character versions of `strrchr`. The arguments and return value of `wcsrchr` are wide-character strings; those of `_mbsrchr` are multibyte-character strings.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload taking a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CRT_CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

`_mbsrchr` validates its parameters. If *str* is NULL, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, `errno` is set to EINVAL and `_mbsrchr` returns 0. `strrchr` and `wcsrchr` do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the LC_CTYPE category setting of the locale; for more information, see setlocale. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tcsrchr`	`strrchr`	`_mbsrchr`	`wcsrchr`
n/a	**n/a**	`_mbsrchr_l`	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
`strrchr`	<string.h>
`wcsrchr`	<string.h> or <wchar.h>
`_mbsrchr`, `_mbsrchr_l`	<mbstring.h>

For more information about compatibility, see Compatibility.

Example

For an example of using `strrchr`, see strchr.

See also

String Manipulation
Locale

Interpretation of Multibyte-Character Sequences
strchr, wcschr, _mbschr, _mbschr_l
strcspn, wcscspn, _mbscspn, _mbscspn_l
_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strrev, wcsrev

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strrev, _wcsrev, _mbsrev, _mbsrev_l instead.

_strrev, _wcsrev, _mbsrev, _mbsrev_l

3/1/2019 • 2 minutes to read • Edit Online

Reverses the characters of a string.

> **IMPORTANT**
> **_mbsrev** and **_mbsrev_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strrev(
   char *str
);
wchar_t *_wcsrev(
   wchar_t *str
);
unsigned char *_mbsrev(
   unsigned char *str
);
unsigned char *_mbsrev_l(
   unsigned char *str,
   _locale_t locale
);
```

Parameters

str
Null-terminated string to reverse.

locale
Locale to use.

Return Value

Returns a pointer to the altered string. No return value is reserved to indicate an error.

Remarks

The **_strrev** function reverses the order of the characters in *str*. The terminating null character remains in place. **_wcsrev** and **_mbsrev** are wide-character and multibyte-character versions of **_strrev**. The arguments and return value of **_wcsrev** are wide-character strings; those of **_mbsrev** are multibyte-character strings. For **_mbsrev**, the order of bytes in each multibyte character in *str* is not changed. These three functions behave identically otherwise.

_mbsrev validates its parameters. If either *string1* or *string2* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbsrev** returns **NULL** and sets **errno** to **EINVAL**. **_strrev** and **_wcsrev** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions are identical, except that the ones that don't have the **_l** suffix use the current locale and the ones that do have the **_l** suffix instead use the locale parameter

that's passed in. For more information, see Locale.

> **IMPORTANT**
> These functions might be vulnerable to buffer overrun threats. Buffer overruns can be used for system attacks because they can cause an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsrev	**_strrev**	**_mbsrev**	**_wcsrev**
n/a	**n/a**	**_mbsrev_l**	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
_strrev	<string.h>
_wcsrev	<string.h> or <wchar.h>
_mbsrev, **_mbsrev_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strrev.c
// This program checks a string to see
// whether it is a palindrome: that is, whether
// it reads the same forward and backward.
//

#include <string.h>
#include <stdio.h>

int main( void )
{
   char* string = "Able was I ere I saw Elba";
   int result;

   // Reverse string and compare (ignore case):
   result = _stricmp( string, _strrev( _strdup( string ) ) );
   if( result == 0 )
      printf( "The string \"%s\" is a palindrome\n", string );
   else
      printf( "The string \"%s\" is not a palindrome\n", string );
}
```

```
The string "Able was I ere I saw Elba" is a palindrome
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcpy, wcscpy, _mbscpy
_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

strset, wcsset

10/31/2018 • 2 minutes to read • Edit Online

These functions are deprecated. Use the ISO C++ conformant _strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l or security-enhanced _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l instead.

_strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l

3/1/2019 • 2 minutes to read • Edit Online

Sets characters of a string to a character. More secure versions of these functions are available; see _strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l.

> **IMPORTANT**
>
> **_mbsset** and **_mbsset_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strset(
   char *str,
   int c
);
char *_strset_l(
   char *str,
   int c,
   locale_t locale
);
wchar_t *_wcsset(
   wchar_t *str,
   wchar_t c
);
wchar_t *_wcsset_l(
   wchar_t *str,
   wchar_t c,
   locale_t locale
);
unsigned char *_mbsset(
   unsigned char *str,
   unsigned int c
);
unsigned char *_mbsset_l(
   unsigned char *str,
   unsigned int c,
   _locale_t locale
);
```

Parameters

str
Null-terminated string to be set.

c
Character setting.

locale
Locale to use.

Return Value

Returns a pointer to the altered string.

Remarks

The **_strset** function sets all characters (except the terminating null character) of *str* to *c*, converted to **char**. **_wcsset** and **_mbsset_l** are wide-character and multibyte-character versions of **_strset**, and the data types of the arguments and return values vary accordingly. These functions behave identically otherwise.

_mbsset validates its parameters. If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_mbsset** returns **NULL** and sets **errno** to **EINVAL**. **_strset** and **_wcsset** do not validate their parameters.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale, _wsetlocale for more information. The versions of these functions are identical, except that the ones that don't have the **_l** suffix use the current locale and the ones that do have the **_l** suffix instead use the locale parameter that's passed in. For more information, see Locale.

> **IMPORTANT**
> These functions might be vulnerable to buffer overrun threats. Buffer overruns can be used for system attacks because they can cause an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsset	**_strset**	**_mbsset**	**_wcsset**
_tcsset_l	**_strset_l**	**_mbsset_l**	**_wcsset_l**

Requirements

ROUTINE	REQUIRED HEADER
_strset	<string.h>
_strset_l	<tchar.h>
_wcsset	<string.h> or <wchar.h>
_wcsset_l	<tchar.h>
_mbsset, **_mbsset_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strset.c
// compile with: /W3

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[] = "Fill the string with something.";
   printf( "Before: %s\n", string );
   _strset( string, '*' ); // C4996
   // Note: _strset is deprecated; consider using _strset_s instead
   printf( "After:  %s\n", string );
}
```

```
Before: Fill the string with something.
After:  ******************************
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbset, _mbsnbset_l
memset, wmemset
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

_strset_s, _strset_s_l, _wcsset_s, _wcsset_s_l, _mbsset_s, _mbsset_s_l

3/1/2019 • 2 minutes to read • Edit Online

Sets characters of a string to a character. These versions of _strset, _strset_l, _wcsset, _wcsset_l, _mbsset, _mbsset_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbsset_s** and **_mbsset_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _strset_s(
   char *str,
   size_t numberOfElements,
   int c
);
errno_t _strset_s_l(
   char *str,
   size_t numberOfElements,
   int c,
   locale_t locale
);
errno_t _wcsset_s(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c
);
errno_t *_wcsset_s_l(
   wchar_t *str,
   size_t numberOfElements,
   wchar_t c,
   locale_t locale
);
errno_t _mbsset_s(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c
);
errno_t _mbsset_s_l(
   unsigned char *str,
   size_t numberOfElements,
   unsigned int c,
   _locale_t locale
);
```

Parameters

str

Null-terminated string to be set.

numberOfElements

The size of the *str* buffer.

c
Character setting.

locale
Locale to use.

Return Value

Zero if successful, otherwise an error code.

These functions validate their arguments. If *str* is a null pointer, or the *numberOfElements* argument is less than or equal to 0, or the block passed in is not null-terminated, then the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EINVAL** and set **errno** to **EINVAL**.

Remarks

The **_strset_s** function sets all the characters of *str* to *c* (converted to **char**), except the terminating null character. **_wcsset_s** and **_mbsset_s** are wide-character and multibyte-character versions of **_strset_s**. The data types of the arguments and return values vary accordingly. These functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsset_s	**_strset_s**	**_mbsset_s**	**_wcsset_s**
_tcsset_s_l	**_strset_s_l**	**_mbsset_s_l**	**_wcsset_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_strset_s	<string.h>
_strset_s_l	<tchar.h>
_wcsset_s	<string.h> or <wchar.h>
_wcsset_s_l	<tchar.h>
_mbsset_s, **_mbsset_s_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strset_s.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char string[] = "Fill the string with something.";
   printf( "Before: %s\n", string );
   _strset_s( string, _countof(string), '*' );
   printf( "After:  %s\n", string );
}
```

```
Before: Fill the string with something.
After:  ******************************
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
_mbsnbset, _mbsnbset_l
memset, wmemset
strcat, wcscat, _mbscat
strcmp, wcscmp, _mbscmp
strcpy, wcscpy, _mbscpy
_strnset, _strnset_l, _wcsnset, _wcsnset_l, _mbsnset, _mbsnset_l

strspn, wcsspn, _mbsspn, _mbsspn_l

3/1/2019 • 2 minutes to read • Edit Online

Returns the index of the first character, in a string, that does not belong to a set of characters.

> **IMPORTANT**
> **_mbsspn** and **_mbsspn_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
size_t strspn(
   const char *str,
   const char *strCharSet
);
size_t wcsspn(
   const wchar_t *str,
   const wchar_t *strCharSet
);
size_t _mbsspn(
   const unsigned char *str,
   const unsigned char *strCharSet
);
size_t _mbsspn_l(
   const unsigned char *str,
   const unsigned char *strCharSet,
   _locale_t locale
);
```

Parameters

str
Null-terminated string to search.

strCharSet
Null-terminated character set.

locale
Locale to use.

Return Value

Returns an integer value specifying the length of the substring in *str* that consists entirely of characters in *strCharSet*. If *str* begins with a character not in *strCharSet*, the function returns 0.

Remarks

The **strspn** function returns the index of the first character in *str* that does not belong to the set of characters in *strCharSet*. The search does not include terminating null characters.

wcsspn and **_mbsspn** are wide-character and multibyte-character versions of **strspn**. The arguments of **wcsspn** are wide-character strings; those of **_mbsspn** are multibyte-character strings. **_mbsspn** validates its parameters. If *str* or *strCharSet* is **NULL**, the invalid parameter handler is invoked, as described in Parameter

Validation . If execution is allowed to continue, **_mbspn** sets **errno** to **EINVAL** and returns 0. **strspn** and **wcsspn** do not validate their parameters. These three functions behave identically otherwise.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale; see setlocale for more information. The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior; the versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsspn	**strspn**	**_mbsspn**	**wcsspn**
n/a	**n/a**	**_mbsspn_l**	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
strspn	<string.h>
wcsspn	<string.h> or <wchar.h>
_mbsspn, **_mbsspn_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strspn.c
// This program uses strspn to determine
// the length of the segment in the string "cabbage"
// consisting of a's, b's, and c's. In other words,
// it finds the first non-abc letter.
//

#include <string.h>
#include <stdio.h>

int main( void )
{
   char string[] = "cabbage";
   int  result;
   result = strspn( string, "abc" );
   printf( "The portion of '%s' containing only a, b, or c "
           "is %d bytes long\n", string, result );
}
```

```
The portion of 'cabbage' containing only a, b, or c is 5 bytes long
```

See also

String Manipulation

Locale

Interpretation of Multibyte-Character Sequences

_strspnp, _wcsspnp, _mbsspnp, _mbsspnp_l

strcspn, wcscspn, _mbscspn, _mbscspn_l

strncat, _strncat_l, wcsncat, _wcsncat_l, _mbsncat, _mbsncat_l

strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

strncpy, _strncpy_l, wcsncpy, _wcsncpy_l, _mbsncpy, _mbsncpy_l

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l

strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

_strspnp, _wcsspnp, _mbsspnp, _mbsspnp_l

11/9/2018 • 2 minutes to read • Edit Online

Returns a pointer to the first character in a given string that is not in another given string.

> **IMPORTANT**
> **_mbsspnp** and **_mbsspnp_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strspnp(
   const char *str,
   const char *charset
);
wchar_t *_wcsspnp(
   const unsigned wchar_t *str,
   const unsigned wchar_t *charset
);
unsigned char *_mbsspnp(
   const unsigned char *str,
   const unsigned char *charset
);
unsigned char *_mbsspnp_l(
   const unsigned char *str,
   const unsigned char *charset,
   _locale_t locale
);
```

Parameters

str
Null-terminated string to search.

charset
Null-terminated character set.

locale
Locale to use.

Return Value

_strspnp, **_wcsspnp**, and **_mbsspnp** return a pointer to the first character in *str* that does not belong to the set of characters in *charset*. Each of these functions returns **NULL** if *str* consists entirely of characters from *charset*. For each of these routines, no return value is reserved to indicate an error.

Remarks

The **_mbsspnp** function returns a pointer to the multibyte character that is the first character in *str* that does not belong to the set of characters in *charset*. **_mbsspnp** recognizes multibyte-character sequences according to the multibyte code page currently in use. The search does not include terminating null characters.

If either *str* or *charset* is a null pointer, this function invokes the invalid parameter handler, as described in

Parameter Validation. If execution is allowed to continue, the function returns **NULL** and sets **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsspnp	**_strspnp**	**_mbsspnp**	**_wcsspnp**

_strspnp and **_wcsspnp** are single-byte character and wide-character versions of **_mbsspnp**. **_strspnp** and **_wcsspnp** behave identically to **_mbsspnp** otherwise; they are provided only for this mapping and should not be used for any other reason. For more information, see Using Generic-Text Mappings and Generic-Text Mappings.

_mbsspnp_l is identical except that it uses the locale parameter passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
_mbsspnp	<mbstring.h>
_strspnp	<tchar.h>
_wcsspnp	<tchar.h>

For more compatibility information, see Compatibility.

Example

```
// crt_mbsspnp.c
#include <mbstring.h>
#include <stdio.h>

int main( void ) {
   const unsigned char string1[] = "cabbage";
   const unsigned char string2[] = "c";
   unsigned char *ptr = 0;
   ptr = _mbsspnp( string1, string2 );
   printf( "%s\n", ptr);
}
```

Output

```
abbage
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strspn, wcsspn, _mbsspn, _mbsspn_l
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l

_strnicmp, _wcsnicmp, _mbsnicmp, _strnicmp_l, _wcsnicmp_l, _mbsnicmp_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l

strstr, wcsstr, _mbsstr, _mbsstr_l

10/31/2018 • 3 minutes to read • Edit Online

Returns a pointer to the first occurrence of a search string in a string.

IMPORTANT

`_mbsstr` and `_mbsstr_l` cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strstr(
   const char *str,
   const char *strSearch
); // C only
char *strstr(
   char *str,
   const char *strSearch
); // C++ only
const char *strstr(
   const char *str,
   const char *strSearch
); // C++ only
wchar_t *wcsstr(
   const wchar_t *str,
   const wchar_t *strSearch
); // C only
wchar_t *wcsstr(
   wchar_t *str,
   const wchar_t *strSearch
); // C++ only
const wchar_t *wcsstr(
   const wchar_t *str,
   const wchar_t *strSearch
); // C++ only
unsigned char *_mbsstr(
   const unsigned char *str,
   const unsigned char *strSearch
); // C only
unsigned char *_mbsstr(
   unsigned char *str,
   const unsigned char *strSearch
); // C++ only
const unsigned char *_mbsstr(
   const unsigned char *str,
   const unsigned char *strSearch
); // C++ only
unsigned char *_mbsstr_l(
   const unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C only
unsigned char *_mbsstr_l(
   unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C++ only
const unsigned char *_mbsstr_l(
   const unsigned char *str,
   const unsigned char *strSearch,
   _locale_t locale
); // C++ only
```

Parameters

str
Null-terminated string to search.

strSearch
Null-terminated string to search for.

locale
Locale to use.

Return Value

Returns a pointer to the first occurrence of *strSearch* in *str*, or NULL if *strSearch* does not appear in *str*. If *strSearch* points to a string of zero length, the function returns *str*.

Remarks

The `strstr` function returns a pointer to the first occurrence of *strSearch* in *str*. The search does not include terminating null characters. `wcsstr` is the wide-character version of `strstr` and `_mbsstr` is the multibyte-character version. The arguments and return value of `wcsstr` are wide-character strings; those of `_mbsstr` are multibyte-character strings. `_mbsstr` validates its parameters. If *str* or *strSearch* is NULL, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, `_mbsstr` sets `errno` to EINVAL and returns 0. `strstr` and `wcsstr` do not validate their parameters. These three functions behave identically otherwise.

> **IMPORTANT**
> These functions might incur a threat from a buffer overrun problem. Buffer overrun problems can be used to attack a system because they can allow the execution of arbitrary code, which can cause an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

In C, these functions take a **const** pointer for the first argument. In C++, two overloads are available. The overload that takes a pointer to **const** returns a pointer to **const**; the version that takes a pointer to non-**const** returns a pointer to non-**const**. The macro _CRT_CONST_CORRECT_OVERLOADS is defined if both the **const** and non-**const** versions of these functions are available. If you require the non-**const** behavior for both C++ overloads, define the symbol _CONST_RETURN.

The output value is affected by the locale-category setting of LC_CTYPE; for more information, see setlocale, _wsetlocale. The versions of these functions that do not have the **_l** suffix use the current locale for this locale-dependent behavior; the versions that have the **_l** suffix are identical except that they instead use the locale parameter that's passed in. For more information, see Locale.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
`_tcsstr`	`strstr`	`_mbsstr`	`wcsstr`
n/a	**n/a**	`_mbsstr_l`	**n/a**

Requirements

ROUTINE	REQUIRED HEADER
`strstr`	<string.h>
`wcsstr`	<string.h> or <wchar.h>
`_mbsstr`, `_mbsstr_l`	<mbstring.h>

For more information about compatibility, see Compatibility.

Example

```
// crt_strstr.c

#include <string.h>
#include <stdio.h>

char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

int main( void )
{
   char *pdest;
   int  result;
   printf( "String to be searched:\n   %s\n", string );
   printf( "   %s\n   %s\n\n", fmt1, fmt2 );
   pdest = strstr( string, str );
   result = (int)(pdest - string + 1);
   if ( pdest != NULL )
      printf( "%s found at position %d\n", str, result );
   else
      printf( "%s not found\n", str );
}
```

```
String to be searched:
   The quick brown dog jumps over the lazy fox
            1         2         3         4         5
   12345678901234567890123456789012345678901234567890

lazy found at position 36
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strcmp, wcscmp, _mbscmp
strpbrk, wcspbrk, _mbspbrk, _mbspbrk_l
strrchr, wcsrchr, _mbsrchr, _mbsrchr_l
strspn, wcsspn, _mbsspn, _mbsspn_l
basic_string::find

_strtime, _wstrtime

10/31/2018 • 2 minutes to read • Edit Online

Copy the time to a buffer. More secure versions of these functions are available; see _strtime_s, _wstrtime_s.

Syntax

```
char *_strtime(
   char *timestr
);
wchar_t *_wstrtime(
   wchar_t *timestr
);
template <size_t size>
char *_strtime(
   char (&timestr)[size]
); // C++ only
template <size_t size>
wchar_t *_wstrtime(
   wchar_t (&timestr)[size]
); // C++ only
```

Parameters

timestr
Time string.

Return Value

Returns a pointer to the resulting character string *timestr*.

Remarks

The **_strtime** function copies the current local time into the buffer pointed to by *timestr*. The time is formatted as **hh:mm:ss** where **hh** is two digits representing the hour in 24-hour notation, **mm** is two digits representing the minutes past the hour, and **ss** is two digits representing seconds. For example, the string **18:23:44** represents 23 minutes and 44 seconds past 6 P.M. The buffer must be at least 9 bytes long.

_wstrtime is a wide-character version of **_strtime**; the argument and return value of **_wstrtime** are wide-character strings. These functions behave identically otherwise. If *timestr* is a **NULL** pointer or if *timestr* is formatted incorrectly, the invalid parameter handler is invoked, as described in Parameter Validation. If the exception is allowed to continue, these functions return a **NULL** and set **errno** to **EINVAL** if *timestr* was a **NULL** or set **errno** to **ERANGE** if *timestr* is formatted incorrectly.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstrtime	**_strtime**	**_strtime**	**_wstrtime**

Requirements

ROUTINE	REQUIRED HEADER
_strtime	<time.h>
_wstrtime	<time.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtime.c
// compile with: /W3

#include <time.h>
#include <stdio.h>

int main( void )
{
   char tbuffer [9];
   _strtime( tbuffer ); // C4996
   // Note: _strtime is deprecated; consider using _strtime_s instead
   printf( "The current time is %s \n", tbuffer );
}
```

```
The current time is 14:21:44
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

_strtime_s, _wstrtime_s

10/31/2018 • 2 minutes to read • Edit Online

Copy the current time to a buffer. These are versions of _strtime, _wstrtime with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _strtime_s(
   char *buffer,
   size_t numberOfElements
);
errno_t _wstrtime_s(
   wchar_t *buffer,
   size_t numberOfElements
);
template <size_t size>
errno_t _strtime_s(
   char (&buffer)[size]
); // C++ only
template <size_t size>
errno_t _wstrtime_s(
   wchar_t (&buffer)[size]
); // C++ only
```

Parameters

buffer
A buffer, at least 10 bytes long, where the time will be written.

numberOfElements
The size of the buffer.

Return Value

Zero if successful.

If an error condition occurs, the invalid parameter handler is invoked, as described in Parameter Validation. The return value is an error code if there is a failure. Error codes are defined in ERRNO.H; see the following table for the exact errors generated by this function. For more information on error codes, see errno Constants.

Error Conditions

BUFFER	*NUMBEROFELEMENTS*	RETURN	CONTENTS OF *BUFFER*
NULL	(any)	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0	**EINVAL**	Not modified
Not **NULL** (pointing to valid buffer)	0 < size < 9	**EINVAL**	Empty string

BUFFER	*NUMBEROFELEMENTS*	RETURN	CONTENTS OF *BUFFER*
Not **NULL** (pointing to valid buffer)	Size > 9	0	Current time formatted as specified in the remarks

Security Issues

Passing in an invalid non-**NULL** value for the buffer will result in an access violation if the *numberOfElements* parameter is greater than 9.

Passing a value for *numberOfElements* that is greater than the actual size of the buffer will result in buffer overrun.

Remarks

These functions provide more secure versions of _strtime and _wstrtime. The **_strtime_s** function copies the current local time into the buffer pointed to by *timestr*. The time is formatted as **hh:mm:ss** where **hh** is two digits representing the hour in 24-hour notation, **mm** is two digits representing the minutes past the hour, and **ss** is two digits representing seconds. For example, the string **18:23:44** represents 23 minutes and 44 seconds past 6 P.M. The buffer must be at least 9 bytes long; the actual size is specified by the second parameter.

_wstrtime is a wide-character version of **_strtime**; the argument and return value of **_wstrtime** are wide-character strings. These functions behave identically otherwise.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mapping:

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tstrtime_s	**_strtime_s**	**_strtime_s**	**_wstrtime_s**

Requirements

ROUTINE	REQUIRED HEADER
_strtime_s	<time.h>
_wstrtime_s	<time.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// strtime_s.c

#include <time.h>
#include <stdio.h>

int main()
{
    char tmpbuf[9];
    errno_t err;

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    // Display operating system-style date and time.
    err = _strtime_s( tmpbuf, 9 );
    if (err)
    {
       printf("_strdate_s failed due to an invalid argument.");
      exit(1);
    }
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    err = _strdate_s( tmpbuf, 9 );
    if (err)
    {
       printf("_strdate_s failed due to an invalid argument.");
       exit(1);
    }
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );

}
```

```
OS time:            14:37:49
OS date:            04/25/03
```

See also

Time Management
asctime_s, _wasctime_s
ctime_s, _ctime32_s, _ctime64_s, _wctime_s, _wctime32_s, _wctime64_s
gmtime_s, _gmtime32_s, _gmtime64_s
localtime_s, _localtime32_s, _localtime64_s
mktime, _mktime32, _mktime64
time, _time32, _time64
_tzset

strtod, _strtod_l, wcstod, _wcstod_l

10/31/2018 • 4 minutes to read • Edit Online

Convert strings to a double-precision value.

Syntax

```
double strtod(
   const char *strSource,
   char **endptr
);
double _strtod_l(
   const char *strSource,
   char **endptr,
   _locale_t locale
);
double wcstod(
   const wchar_t *strSource,
   wchar_t **endptr
);
double wcstod_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to character that stops scan.

locale
The locale to use.

Return Value

strtod returns the value of the floating-point number, except when the representation would cause an overflow, in which case the function returns +/-**HUGE_VAL**. The sign of **HUGE_VAL** matches the sign of the value that cannot be represented. **strtod** returns 0 if no conversion can be performed or an underflow occurs.

wcstod returns values analogously to **strtod**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs and the invalid parameter handler is invoked, as described in Parameter Validation. See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this and other return codes.

Remarks

Each function converts the input string *strSource* to a **double**. The **strtod** function converts *strSource* to a double-precision value. **strtod** stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character. **wcstod** is a wide-character version of **strtod**; its *strSource* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstod	**strtod**	**strtod**	**wcstod**
_tcstod_l	**_strtod_l**	**_strtod_l**	**_wcstod_l**

The **LC_NUMERIC** category setting of the current locale determines recognition of the radix point character in *strSource*. For more information, see setlocale. The functions without the **_l** suffix use the current locale; **_strtod_l** is identical to **_strtod_l** except that they use the *locale* passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location pointed to by *endptr*.

strtod expects *strSource* to point to a string of one of the following forms:

[*whitespace*] [*sign*] {*digits* [*radix digits*] | *radix digits*} [{**e** | **E**} [*sign*] *digits*] [*whitespace*] [*sign*] {**0x** | **0X**} {*hexdigits* [*radix hexdigits*] | *radix hexdigits*} [{**p** | **P**} [*sign*] *hexdigits*] [*whitespace*] [*sign*] {**INF** | **INFINITY**} [*whitespace*] [*sign*] **NAN** [*sequence*]

The optional leading *whitespace* may consist of space and tab characters, which are ignored; *sign* is either plus (+) or minus (-); *digits* are one or more decimal digits; *hexdigits* are one or more hexadecimal digits; *radix* is the radix point character, either a period (.) in the default "C" locale, or the locale-specific value if the current locale is different or when *locale* is specified; a *sequence* is a sequence of alphanumeric or underscore characters. In both decimal and hexadecimal number forms, if no digits appear before the radix point character, at least one must appear after the radix point character. In the decimal form, the decimal digits can be followed by an exponent, which consists of an introductory letter (**e** or **E**) and an optionally signed integer. In the hexadecimal form, the hexadecimal digits can be followed by an exponent, which consists of an introductory letter (**p** or **P**) and an optionally signed hexadecimal integer that represents the exponent as a power of 2. In either form, if neither an exponent part nor a radix point character appears, a radix point character is assumed to follow the last digit in the string. Case is ignored in both the **INF** and **NAN** forms. The first character that does not fit one of these forms stops the scan.

The UCRT versions of these functions do not support conversion of Fortran-style (**d** or **D**) exponent letters. This non-standard extension was supported by earlier versions of the CRT, and may be a breaking change for your code. The UCRT versions support hexadecimal strings and round-tripping of INF and NAN values, which were not supported in earlier versions. This can also cause breaking changes in your code. For example, the string "0x1a" would be interpreted by **strtod** as 0.0 in previous versions, but as 26.0 in the UCRT version.

Requirements

ROUTINE	REQUIRED HEADER
strtod, **_strtod_l**	C: <stdlib.h> C++: <cstdlib> or <stdlib.h>
wcstod, **_wcstod_l**	C: <stdlib.h> or <wchar.h> C++: <cstdlib>, <stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtod.c
// This program uses strtod to convert a
// string to a double-precision value; strtol to
// convert a string to long integer values; and strtoul
// to convert a string to unsigned long-integer values.
//

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char   *string, *stopstring;
   double x;
   long   l;
   int    base;
   unsigned long ul;

   string = "3.1415926This stopped it";
   x = strtod( string, &stopstring );
   printf( "string = %s\n", string );
   printf("   strtod = %f\n", x );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "-10110134932This stopped it";
   l = strtol( string, &stopstring, 10 );
   printf( "string = %s\n", string );
   printf("   strtol = %ld\n", l );
   printf("   Stopped scan at: %s\n\n", stopstring );

   string = "10110134932";
   printf( "string = %s\n", string );

   // Convert string using base 2, 4, and 8:
   for( base = 2; base <= 8; base *= 2 )
   {
      // Convert the string:
      ul = strtoul( string, &stopstring, base );
      printf( "   strtol = %ld (base %d)\n", ul, base );
      printf( "   Stopped scan at: %s\n", stopstring );
   }
}
```

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at: This stopped it

string = -10110134932This stopped it
   strtol = -2147483648
   Stopped scan at: This stopped it

string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at: 34932
   strtol = 4423 (base 4)
   Stopped scan at: 4932
   strtol = 2134108 (base 8)
   Stopped scan at: 932
```

See also

Data Conversion
Floating-Point Support

Interpretation of Multibyte-Character Sequences
Locale
String to Numeric Value Functions
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l
localeconv
_create_locale, _wcreate_locale
_free_locale

strtof, _strtof_l, wcstof, _wcstof_l

10/31/2018 • 3 minutes to read • Edit Online

Converts strings to a single-precision floating-point value.

Syntax

```
float strtof(
   const char *strSource,
   char **endptr
);
float _strtof_l(
   const char *strSource,
   char **endptr,
   _locale_t locale
);
float wcstof(
   const wchar_t *strSource,
   wchar_t **endptr
);
float wcstof_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

locale
The locale to use.

Return Value

strtof returns the value of the floating-point number, except when the representation would cause an overflow, in which case the function returns +/-**HUGE_VALF**. The sign of **HUGE_VALF** matches the sign of the value that cannot be represented. **strtof** returns 0 if no conversion can be performed or an underflow occurs.

wcstof returns values analogously to **strtof**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs and the invalid parameter handler is invoked, as described in Parameter Validation.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each function converts the input string *strSource* to a **float**. The **strtof** function converts *strSource* to a single-precision value. **strtof** stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character. **wcstof** is a wide-character version of **strtof**; its *strSource* argument is a wide-character string. Otherwise, these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstof	**strtof**	**strtof**	**wcstof**
_tcstof_l	**_strtof_l**	**_strtof_l**	**_wcstof_l**

The **LC_NUMERIC** category setting of the current locale determines recognition of the radix character in *strSource*; for more information, see setlocale, _wsetlocale. The functions that don't have the **_l** suffix use the current locale; the ones that have the suffix are identical except that they use the locale that's passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

strtof expects *strSource* to point to a string of the following form:

[*whitespace*] [*sign*] [*digits*] [**.***digits*] [{**e** | **E**} [*sign*] *digits*]

A *whitespace* may consist of space and tab characters, which are ignored; *sign* is either plus (**+**) or minus (**-**); and *digits* are one or more decimal digits. If no digits appear before the radix character, at least one must appear after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (**e** or **E**) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

The UCRT versions of these functions do not support conversion of Fortran-style (**d** or **D**) exponent letters. This non-standard extension was supported by earlier versions of the CRT, and may be a breaking change for your code.

Requirements

ROUTINE	REQUIRED HEADER
strtof, **_strtof_l**	C: <stdlib.h> C++: <cstdlib> or <stdlib.h>
wcstof, **_wcstof_l**	C: <stdlib.h> or <wchar.h> C++: <cstdlib>, <stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtof.c
// This program uses strtof to convert a
// string to a single-precision value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *string;
   char *stopstring;
   float x;

   string = "3.14159This stopped it";
   x = strtof(string, &stopstring);
   printf("string = %s\n", string);
   printf("   strtof = %f\n", x);
   printf("   Stopped scan at: %s\n\n", stopstring);
}
```

```
string = 3.14159This stopped it
   strtof = 3.141590
   Stopped scan at: This stopped it
```

See also

Data Conversion
Floating-Point Support
Interpretation of Multibyte-Character Sequences
Locale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l
localeconv
_create_locale, _wcreate_locale
_free_locale

_strtoi64, _wcstoi64, _strtoi64_l, _wcstoi64_l

11/8/2018 • 3 minutes to read • Edit Online

Convert a string to an **__int64** value.

Syntax

```
__int64 _strtoi64(
   const char *strSource,
   char **endptr,
   int base
);
__int64 _wcstoi64(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
__int64 _strtoi64_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
__int64 _wcstoi64_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to character that stops scan.

base
Number base to use.

locale
The locale to use.

Return Value

_strtoi64 returns the value represented in the string *strSource*, except when the representation would cause an overflow, in which case it returns **_I64_MAX** or **_I64_MIN**. The function will return 0 if no conversion can be performed. **_wcstoi64** returns values analogously to **strtoi64**.

_I64_MAX and **_I64_MIN** are defined in LIMITS.H.

If *strSource* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_strtoi64** function converts *strSource* to an **__int64**. Both functions stop reading the string *strSource* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*. **_wcstoi64** is a wide-character version of **_strtoi64**; its *strSource* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoi64	**_strtoi64**	**_strtoi64**	**_wcstoi64**
_tcstoi64_l	**_strtoi64_l**	**_strtoi64_l**	**_wcstoi64_l**

The locale's **LC_NUMERIC** category setting determines recognition of the radix character in *strSource*; for more information, see setlocale. The functions without the _l suffix use the current locale; **_strtoi64_l** and **_wcstoi64_l** are identical to the corresponding function without the **_l** suffix except that they use the locale passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location pointed to by *endptr*.

_strtoi64 expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits; *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *strSource* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character will stop the scan.

Requirements

ROUTINE	REQUIRED HEADER
_strtoi64, **_strtoi64_l**	<stdlib.h>
_wcstoi64, **_wcstoi64_l**	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
Locale

localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strtoimax, _strtoimax_l, wcstoimax, _wcstoimax_l

10/31/2018 • 3 minutes to read • Edit Online

Converts a string to an integer value of the largest supported signed integer type.

Syntax

```
intmax_t strtoimax(
   const char *strSource,
   char **endptr,
   int base
);
intmax_t wcstoimax(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
intmax_t _strtoimax_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
intmax_t _wcstoimax_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

base
Number base to use.

locale
The locale to use.

Return Value

strtoimax returns the value that's represented in the string *strSource*, except when the representation would cause an overflow—in that case, it returns **INTMAX_MAX** or **INTMAX_MIN**, and **errno** is set to **ERANGE**. The function returns 0 if no conversion can be performed. **wcstoimax** returns values analogously to **strtoimax**.

INTMAX_MAX and **INTMAX_MIN** are defined in stdint.h.

If *strSource* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **strtoimax** function converts *strSource* to an **intmax_t**. The wide-character version of **strtoimax** is **wcstoimax**; its *strSource* argument is a wide-character string. Otherwise, these functions behave identically. Both functions stop reading the string *strSource* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character that's greater than or equal to *base*.

The locale's **LC_NUMERIC** category setting determines recognition of the radix character in *strSource*; for more information, see setlocale, _wsetlocale. The functions that don't have the **_l** suffix use the current locale; **_strtoimax_l** and **_wcstoimax_l** are identical to the corresponding functions that don't have the **_l** suffix except that they instead use the locale that's passed in. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoimax	**strtoimax**	**strtoimax**	**wcstoimax**
_tcstoimax_l	**strtoimax_l**	**_strtoimax_l**	**_wcstoimax_l**

strtoimax expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits; *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *strSource* are used to determine the base. If the first character is '0' and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character would stop the scan.

Requirements

ROUTINE	REQUIRED HEADER
strtoimax, **_strtoimax_l**, **wcstoimax**, **_wcstoimax_l**	<inttypes.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale

String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
strtoumax, _strtoumax_l, wcstoumax, _wcstoumax_l
atof, _atof_l, _wtof, _wtof_l

strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l

3/26/2019 • 3 minutes to read • Edit Online

Finds the next token in a string, by using the current locale or a specified locale that's passed in. More secure versions of these functions are available; see strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l.

> **IMPORTANT**
>
> **_mbstok** and **_mbstok_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *strtok(
   char *strToken,
   const char *strDelimit
);
char *strtok_l(
   char *strToken,
   const char *strDelimit,
   _locale_t locale
);
wchar_t *wcstok(
   wchar_t *strToken,
   const wchar_t *strDelimit
);
wchar_t *wcstok_l(
   wchar_t *strToken,
   const wchar_t *strDelimit,
   _locale_t locale
);
unsigned char *_mbstok(
   unsigned char *strToken,
   const unsigned char *strDelimit
);
unsigned char *_mbstok_l(
   unsigned char *strToken,
   const unsigned char *strDelimit,
   _locale_t locale
);
```

Parameters

strToken
String containing token or tokens.

strDelimit
Set of delimiter characters.

locale
Locale to use.

Return Value

Returns a pointer to the next token found in *strToken*. The functions return **NULL** when no more tokens are found. Each call modifies *strToken* by substituting a null character for the first delimiter that occurs after the returned token.

Remarks

The **strtok** function finds the next token in *strToken*. The set of characters in *strDelimit* specifies possible delimiters of the token to be found in *strToken* on the current call. **wcstok** and **_mbstok** are wide-character and multibyte-character versions of **strtok**. The arguments and return value of **wcstok** are wide-character strings; those of **_mbstok** are multibyte-character strings. These three functions behave identically otherwise.

> **IMPORTANT**
>
> These functions incur a potential threat brought about by a buffer overrun problem. Buffer overrun problems are a frequent method of system attack, resulting in an unwarranted elevation of privilege. For more information, see Avoiding Buffer Overruns.

On the first call to **strtok**, the function skips leading delimiters and returns a pointer to the first token in *strToken*, terminating the token with a null character. More tokens can be broken out of the remainder of *strToken* by a series of calls to **strtok**. Each call to **strtok** modifies *strToken* by inserting a null character after the **token** returned by that call. To read the next token from *strToken*, call **strtok** with a **NULL** value for the *strToken* argument. The **NULL** *strToken* argument causes **strtok** to search for the next token in the modified *strToken*. The *strDelimit* argument can take any value from one call to the next so that the set of delimiters may vary.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale. For more information, see setlocale.

The versions of these functions without the **_l** suffix use the current locale for this locale-dependent behavior. The versions with the **_l** suffix are identical except that they use the locale parameter passed in instead. For more information, see Locale.

> **NOTE**
>
> Each function uses a thread-local static variable for parsing the string into tokens. Therefore, multiple threads can simultaneously call these functions without undesirable effects. However, within a single thread, interleaving calls to one of these functions is highly likely to produce data corruption and inaccurate results. When parsing different strings, finish parsing one string before starting to parse the next. Also, be aware of the potential for danger when calling one of these functions from within a loop where another function is called. If the other function ends up using one of these functions, an interleaved sequence of calls will result, triggering data corruption.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstok	**strtok**	**_mbstok**	**wcstok**
_tcstok	**_strtok_l**	**_mbstok_l**	**_wcstok_l**

Requirements

ROUTINE	REQUIRED HEADER
strtok	<string.h>
wcstok	<string.h> or <wchar.h>
_mbstok, **_mbstok_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtok.c
// compile with: /W3
// In this program, a loop uses strtok
// to print all the tokens (separated by commas
// or blanks) in the string named "string".
//
#include <string.h>
#include <stdio.h>

char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";
char *token;

int main( void )
{
   printf( "Tokens:\n" );

   // Establish string and get the first token:
   token = strtok( string, seps ); // C4996
   // Note: strtok is deprecated; consider using strtok_s instead
   while( token != NULL )
   {
      // While there are tokens in "string"
      printf( " %s\n", token );

      // Get next token:
      token = strtok( NULL, seps ); // C4996
   }
}
```

```
Tokens:
A
string
of
tokens
and
some
more
tokens
```

See also

String Manipulation
Locale
Interpretation of Multibyte-Character Sequences
strcspn, wcscspn, _mbscspn, _mbscspn_l
strspn, wcsspn, _mbsspn, _mbsspn_l

strtok_s, _strtok_s_l, wcstok_s, _wcstok_s_l, _mbstok_s, _mbstok_s_l

3/26/2019 • 4 minutes to read • Edit Online

Finds the next token in a string, by using the current locale or a locale that's passed in. These versions of strtok, _strtok_l, wcstok, _wcstok_l, _mbstok, _mbstok_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbstok_s** and **_mbstok_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char* strtok_s(
   char* str,
   const char* delimiters,
   char** context
);

char* _strtok_s_l(
   char* str,
   const char* delimiters,
   char** context,
   _locale_t locale
);

wchar_t* wcstok_s(
   wchar_t* str,
   const wchar_t* delimiters,
   wchar_t** context
);

wchar_t *_wcstok_s_l(
   wchar_t* str,
   const wchar_t* delimiters,
   wchar_t** context,
   _locale_t locale
);

unsigned char* _mbstok_s(
   unsigned char* str,
   const unsigned char* delimiters,
   char** context
);

unsigned char* _mbstok_s_l(
   unsigned char* str,
   const unsigned char* delimiters,
   char** context,
   _locale_t locale
);
```

Parameters

str
A string containing the token or tokens to find.

delimiters
The set of delimiter characters to use.

context
Used to store position information between calls to the function.

locale
The locale to use.

Return Value

Returns a pointer to the next token found in *str*. Returns **NULL** when no more tokens are found. Each call modifies *str* by substituting a null character for the first delimiter that occurs after the returned token.

Error Conditions

STR	*DELIMITERS*	*CONTEXT*	RETURN VALUE	ERRNO
NULL	any	pointer to a null pointer	**NULL**	**EINVAL**
any	**NULL**	any	**NULL**	**EINVAL**
any	any	**NULL**	**NULL**	**EINVAL**

If *str* is **NULL** but *context* is a pointer to a valid context pointer, there's no error.

Remarks

The **strtok_s** family of functions finds the next token in *str*. The set of characters in *delimiters* specifies possible delimiters of the token to be found in *str* on the current call. **wcstok_s** and **_mbstok_s** are wide-character and multibyte-character versions of **strtok_s**. The arguments and return values of **wcstok_s** and **_wcstok_s_l** are wide-character strings; those of **_mbstok_s** and **_mbstok_s_l** are multibyte-character strings. These functions behave identically otherwise.

This function validates its parameters. When an error condition occurs, as in the Error Conditions table, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **NULL**.

On the first call to **strtok_s**, the function skips leading delimiters and returns a pointer to the first token in *str*, terminating the token with a null character. More tokens can be broken out of the remainder of *str* by a series of calls to **strtok_s**. Each call to **strtok_s** modifies *str* by inserting a null character after the token returned by that call. The *context* pointer keeps track of which string is being read and where in the string the next token is to be read. To read the next token from *str*, call **strtok_s** with a **NULL** value for the *str* argument, and pass the same *context* parameter. The **NULL** *str* argument causes **strtok_s** to search for the next token in the modified *str*. The *delimiters* argument can take any value from one call to the next so that the set of delimiters may vary.

Since the *context* parameter supersedes the static buffers used in **strtok** and **_strtok_l**, it's possible to parse two strings simultaneously in the same thread.

The output value is affected by the setting of the **LC_CTYPE** category setting of the locale. For more information, see setlocale.

The versions of these functions without the **_l** suffix use the current thread locale for this locale-dependent

behavior. The versions with the **_l** suffix are identical except they instead use the locale specified by the *locale* parameter. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
strtok_s	<string.h>
_strtok_s_l	<string.h>
wcstok_s, **_wcstok_s_l**	<string.h> or <wchar.h>
_mbstok_s, **_mbstok_s_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstok_s	**strtok_s**	**_mbstok_s**	**wcstok_s**
_tcstok_s_l	**_strtok_s_l**	**_mbstok_s_l**	**_wcstok_s_l**

Example

```
// crt_strtok_s.c
// In this program, a loop uses strtok_s
// to print all the tokens (separated by commas
// or blanks) in two strings at the same time.

#include <string.h>
#include <stdio.h>

char string1[] =
    "A string\tof ,,tokens\nand some  more tokens";
char string2[] =
    "Another string\n\tparsed at the same time.";
char seps[]   = " ,\t\n";
char *token1 = NULL;
char *token2 = NULL;
char *next_token1 = NULL;
char *next_token2 = NULL;

int main(void)
{
    printf("Tokens:\n");

    // Establish string and get the first token:
    token1 = strtok_s(string1, seps, &next_token1);
    token2 = strtok_s(string2, seps, &next_token2);

    // While there are tokens in "string1" or "string2"
    while ((token1 != NULL) || (token2 != NULL))
    {
        // Get next token:
        if (token1 != NULL)
        {
            printf(" %s\n", token1);
            token1 = strtok_s(NULL, seps, &next_token1);
        }
        if (token2 != NULL)
        {
            printf("        %s\n", token2);
            token2 = strtok_s(NULL, seps, &next_token2);
        }
    }
}
```

```
Tokens:
A
        Another
string
        string
of
        parsed
tokens
        at
and
        the
some
        same
more
        time.
tokens
```

See also

String Manipulation

Locale

Interpretation of Multibyte-Character Sequences

strcspn, wcscspn, _mbscspn, _mbscspn_l

strspn, wcsspn, _mbsspn, _mbsspn_l

strtol, wcstol, _strtol_l, _wcstol_l

11/8/2018 • 3 minutes to read • Edit Online

Convert strings to a long-integer value.

Syntax

```
long strtol(
   const char *strSource,
   char **endptr,
   int base
);
long wcstol(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
long _strtol_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
long _wcstol_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to character that stops scan.

base
Number base to use.

locale
Locale to use.

Return Value

strtol returns the value represented in the string *strSource*, except when the representation would cause an overflow, in which case it returns **LONG_MAX** or **LONG_MIN**. **strtol** returns 0 if no conversion can be performed. **wcstol** returns values analogously to **strtol**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these and other return codes.

Remarks

The **strtol** function converts *strSource* to a **long**. **strtol** stops reading the string *strSource* at the first character it

cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*.

wcstol is a wide-character version of **strtol**; its *strSource* argument is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstol	**strtol**	**strtol**	**wcstol**
_tcstol_l	**_strtol_l**	**_strtol_l**	**_wcstol_l**

The current locale's **LC_NUMERIC** category setting determines recognition of the radix character in *strSource*; for more information, see setlocale. The functions without the **_l** suffix use the current locale; **_strtol_l** and **_wcstol_l** are identical to the corresponding functions without the **_l** suffix except that they use the locale passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location pointed to by *endptr*.

strtol expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits; *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *strSource* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character will stop the scan.

Requirements

ROUTINE	REQUIRED HEADER
strtol	<stdlib.h>
wcstol	<stdlib.h> or <wchar.h>
_strtol_l	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

See the example for strtod.

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strtold, _strtold_l, wcstold, _wcstold_l

11/9/2018 • 3 minutes to read • Edit Online

Converts strings to a long double-precision floating-point value.

Syntax

```
long double strtold(
   const char *strSource,
   char **endptr
);
long double _strtold_l(
   const char *strSource,
   char **endptr,
   _locale_t locale
);
long double wcstold(
   const wchar_t *strSource,
   wchar_t **endptr
);
long double wcstold_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

locale
The locale to use.

Return Value

strtold returns the value of the floating-point number as a **long double**, except when the representation would cause an overflow—in that case, the function returns +/-**HUGE_VALL**. The sign of **HUGE_VALL** matches the sign of the value that cannot be represented. **strtold** returns 0 if no conversion can be performed or an underflow occurs.

wcstold returns values analogously to **strtold**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs and the invalid parameter handler is invoked, as described in Parameter Validation.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each function converts the input string *strSource* to a **long double**. The **strtold** function stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character. The wide-character version of **strtold** is **wcstold**; its *strSource* argument is a wide-character string. Otherwise, these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstold	**strtold**	**strtold**	**wcstold**
_tcstold_l	**_strtold_l**	**_strtold_l**	**_wcstold_l**

The **LC_NUMERIC** category setting of the current locale determines the recognition of the radix character in *strSource*. For more information, see setlocale, _wsetlocale. The functions without the **_l** suffix use the current locale; **_strtold_l** and **_wcstold_l** are identical to **_strtold** and **_wcstold** except that they instead use the locale that's passed in. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

strtold expects *strSource* to point to a string of the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{**d** | **D** | **e** | **E**}[*sign*]*digits*]

A *whitespace* may consist of space and tab characters, which are ignored; *sign* is either plus (**+**) or minus (**-**); and *digits* are one or more decimal digits. If no digits appear before the radix character, at least one must appear after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (**d**, **D**, **e**, or **E**) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

Requirements

ROUTINE	REQUIRED HEADER
strtold, **_strtold_l**	<stdlib.h>
wcstold, **_wcstold_l**	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtold.c
// Build with: cl /W4 /Tc crt_strtold.c
// This program uses strtold to convert a
// string to a long double-precision value.

#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *string;
   char *stopstring;
   long double x;

   string = "3.1415926535898This stopped it";
   x = strtold(string, &stopstring);
   printf("string = %s\n", string);
   printf("   strtold = %.13Lf\n", x);
   printf("   Stopped scan at: %s\n\n", stopstring);
}
```

```
string = 3.1415926535898This stopped it
   strtold = 3.1415926535898
   Stopped scan at: This stopped it
```

See also

Data Conversion
Floating-Point Support
Interpretation of Multibyte-Character Sequences
Locale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l
localeconv
_create_locale, _wcreate_locale
_free_locale

strtoll, _strtoll_l, wcstoll, _wcstoll_l

10/31/2018 • 3 minutes to read • Edit Online

Converts a string to a **long long** value.

Syntax

```
long long strtoll(
   const char *strSource,
   char **endptr,
   int base
);
long long wcstoll(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
long long _strtoll_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
long long _wcstoll_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

base
Number base to use.

locale
The locale to use.

Return Value

strtoll returns the value that's represented in the string *strSource*, except when the representation would cause an overflow—in that case, it returns **LLONG_MAX** or **LLONG_MIN**. The function returns 0 if no conversion can be performed. **wcstoll** returns values analogously to **strtoll**.

LLONG_MAX and **LLONG_MIN** are defined in LIMITS.H.

If *strSource* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **strtoll** function converts *strSource* to a **long long**. Both functions stop reading the string *strSource* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character that's greater than or equal to *base*. **wcstoll** is a wide-character version of **strtoll**; its *strSource* argument is a wide-character string. Otherwise, these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoll	**strtoll**	**strtoll**	**wcstoll**
_tcstoll_l	**_strtoll_l**	**_strtoll_l**	**_wcstoll_l**

The locale's **LC_NUMERIC** category setting determines recognition of the radix character in *strSource*; for more information, see setlocale, _wsetlocale. The functions that don't have the **_l** suffix use the current locale; **_strtoll_l** and **_wcstoll_l** are identical to the corresponding functions that don't have the suffix, except that they instead use the locale that's passed in. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

strtoll expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits; *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string that's pointed to by *strSource* are used to determine the base. If the first character is '0' and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character stops the scan.

Requirements

ROUTINE	REQUIRED HEADER
strtoll, **_strtoll_l**	<stdlib.h>
wcstoll, **_wcstoll_l**	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
Locale

localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

_strtoui64, _wcstoui64, _strtoui64_l, _wcstoui64_l

3/1/2019 • 3 minutes to read • Edit Online

Convert a string to an unsigned **__int64** value.

Syntax

```
unsigned __int64 _strtoui64(
   const char *strSource,
   char **endptr,
   int base
);
unsigned __int64 _wcstoui64(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
unsigned __int64 _strtoui64_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
unsigned __int64 _wcstoui64(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to character that stops scan.

base
Number base to use.

locale
Locale to use.

Return Value

_strtoui64 returns the value represented in the string *strSource*, except when the representation would cause an overflow, in which case it returns **_UI64_MAX**. **_strtoui64** returns 0 if no conversion can be performed.

_UI64_MAX is defined in LIMITS.H.

If *strSource* is **NULL** or the *base* is nonzero and either less than 2 or greater than 36, **errno** is set to **EINVAL**.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_strtoui64** function converts *strSource* to an **unsigned __int64**. **_wcstoui64** is a wide-character version of **_strtoui64**; its *strSource* argument is a wide-character string. Otherwise these functions behave identically.

Both functions stop reading the string *strSource* at the first character they cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoui64	**_strtoui64**	**_strtoui64**	**_wstrtoui64**
_tcstoui64_l	**_strtoui64_l**	**_strtoui64_l**	**_wstrtoui64_l**

The current locale's **LC_NUMERIC** category setting determines recognition of the radix character in *strSource*; for more information, see setlocale. The functions without the _l suffix use the current locale; **_strtoui64_l** and **_wcstoui64_l** are identical to the corresponding functions without the **_l** suffix except that they use the locale passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location pointed to by *endptr*.

_strtoui64 expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored. *digits* are one or more decimal digits. *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *strSource* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character will stop the scan.

Requirements

ROUTINE	REQUIRED HEADER
_strtoui64	<stdlib.h>
_wcstoui64	<stdlib.h> or <wchar.h>
_strtoui64_l	<stdlib.h>
_wcstoui64_l	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_strtoui64.c
#include <stdio.h>

unsigned __int64 atoui64(const char *szUnsignedInt) {
   return _strtoui64(szUnsignedInt, NULL, 10);
}

int main() {
   unsigned __int64 u = atoui64("18446744073709551615");
   printf( "u = %I64u\n", u );
}
```

```
u = 18446744073709551615
```

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
atof, _atof_l, _wtof, _wtof_l

strtoul, _strtoul_l, wcstoul, _wcstoul_l

3/1/2019 • 3 minutes to read • Edit Online

Convert strings to an unsigned long-integer value.

Syntax

```
unsigned long strtoul(
   const char *strSource,
   char **endptr,
   int base
);
unsigned long _strtoul_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
unsigned long wcstoul(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
unsigned long _wcstoul_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to character that stops scan.

base
Number base to use.

locale
Locale to use.

Return Value

strtoul returns the converted value, if any, or **ULONG_MAX** on overflow. **strtoul** returns 0 if no conversion can be performed. **wcstoul** returns values analogously to **strtoul**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

Each of these functions converts the input string *strSource* to an **unsigned long**.

strtoul stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character greater than or equal to *base*. The **LC_NUMERIC** category setting of the locale determines recognition of the radix character in *strSource*; for more information, see setlocale. **strtoul** and **wcstoul** use the current locale; **_strtoul_l** and **_wcstoul_l** are identical except that they use the locale passed in instead. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location pointed to by *endptr*.

wcstoul is a wide-character version of **strtoul**; its *strSource* argument is a wide-character string. Otherwise these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoul	**strtoul**	**strtoul**	**wcstoul**
_tcstoul_l	**strtoul_l**	**_strtoul_l**	**_wcstoul_l**

strtoul expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored. *digits* are one or more decimal digits. *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *strSource* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character will stop the scan. **strtoul** allows a plus (**+**) or minus (**-**) sign prefix; a leading minus sign indicates that the return value is negated.

Requirements

ROUTINE	REQUIRED HEADER
strtoul	<stdlib.h>
wcstoul	<stdlib.h> or <wchar.h>
_strtoul_l	<stdlib.h>
_wcstoul_l	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for strtod.

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
atof, _atof_l, _wtof, _wtof_l

strtoull, _strtoull_l, wcstoull, _wcstoull_l

10/31/2018 • 3 minutes to read • Edit Online

Converts strings to an unsigned long long-integer value.

Syntax

```
unsigned long long strtoull(
   const char *strSource,
   char **endptr,
   int base
);
unsigned long long _strtoull_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
unsigned long long wcstoull(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
unsigned long long _wcstoull_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

base
Number base to use.

locale
Locale to use.

Return Value

strtoull returns the converted value, if any, or **ULLONG_MAX** on overflow. **strtoull** returns 0 if no conversion can be performed. **wcstoull** returns values analogously to **strtoull**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions converts the input string *strSource* to an **unsigned long long** integer value.

strtoull stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character that's greater than or equal to *base*. The setting of the **LC_NUMERIC** category of the locale determines recognition of the radix character in *strSource*; for more information, see setlocale, _wsetlocale. **strtoull** and **wcstoull** use the current locale; **_strtoull_l** and **_wcstoull_l** instead use the locale that's passed in but are identical otherwise. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

wcstoull is a wide-character version of **strtoull** and its *strSource* argument is a wide-character string. Otherwise, these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoull	**strtoull**	**strtoull**	**wcstoull**
_tcstoull_l	**strtoull_l**	**_strtoull_l**	**_wcstoull_l**

strtoull expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored. *digits* are one or more decimal digits. *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string that's pointed to by *strSource* are used to determine the base. If the first character is '0' and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character stops the scan. **strtoull** allows a plus sign (**+**) or minus sign (**-**) prefix; a leading minus sign indicates that the return value is negated.

Requirements

ROUTINE	REQUIRED HEADER
strtoull	<stdlib.h>
wcstoull	<stdlib.h> or <wchar.h>
_strtoull_l	<stdlib.h>
_wcstoull_l	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for strtod.

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
strtoll, _strtoll_l, wcstoll, _wcstoll_l
atof, _atof_l, _wtof, _wtof_l

strtoumax, _strtoumax_l, wcstoumax, _wcstoumax_l

10/31/2018 • 3 minutes to read • Edit Online

Converts strings to an integer value of the largest supported unsigned integer type.

Syntax

```
uintmax_t strtoumax(
   const char *strSource,
   char **endptr,
   int base
);
uintmax_t _strtoumax_l(
   const char *strSource,
   char **endptr,
   int base,
   _locale_t locale
);
uintmax_t wcstoumax(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base
);
uintmax_t _wcstoumax_l(
   const wchar_t *strSource,
   wchar_t **endptr,
   int base,
   _locale_t locale
);
```

Parameters

strSource
Null-terminated string to convert.

endptr
Pointer to the character that stops the scan.

base
Number base to use.

locale
Locale to use.

Return Value

strtoumax returns the converted value, if any, or **UINTMAX_MAX** on overflow. **strtoumax** returns 0 if no conversion can be performed. **wcstoumax** returns values analogously to **strtoumax**. For both functions, **errno** is set to **ERANGE** if overflow or underflow occurs.

For more information about return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions converts the input string *strSource* to a **uintmax_t** integer value.

strtoumax stops reading the string *strSource* at the first character it cannot recognize as part of a number. This may be the terminating null character, or it may be the first numeric character that's greater than or equal to *base*. The **LC_NUMERIC** category setting of the locale determines the recognition of the radix character in *strSource*. For more information, see setlocale, _wsetlocale. **strtoumax** and **wcstoumax** use the current locale; **_strtoumax_l** and **_wcstoumax_l** are identical except that they instead use the locale that's passed in. For more information, see Locale.

If *endptr* is not **NULL**, a pointer to the character that stopped the scan is stored at the location that's pointed to by *endptr*. If no conversion can be performed (no valid digits were found or an invalid base was specified), the value of *strSource* is stored at the location that's pointed to by *endptr*.

The wide-character version of **strtoumax** is **wcstoumax**; its *strSource* argument is a wide-character string. Otherwise, these functions behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcstoumax	**strtoumax**	**strtoumax**	**wcstoumax**
_tcstoumax_l	**strtoumax_l**	**_strtoumax_l**	**_wcstoumax_l**

strtoumax expects *strSource* to point to a string of the following form:

```
[whitespace] [{+ | -}] [0 [{ x | X }]] [digits | letters]
```

A *whitespace* may consist of space and tab characters, which are ignored. *digits* are one or more decimal digits. *letters* are one or more of the letters 'a' through 'z' (or 'A' through 'Z'). The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string that's pointed to by *strSource* are used to determine the base. If the first character is '0' and the second character is not 'x' or 'X', the string is interpreted as an octal integer. If the first character is '0' and the second character is 'x' or 'X', the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted. The first character outside the range of the base stops the scan. For example, if *base* is 0 and the first character scanned is '0', an octal integer is assumed and an '8' or '9' character would stop the scan. **strtoumax** allows a plus sign (**+**) or minus sign (**-**) prefix; a leading minus sign indicates that the return value is the two's complement of the absolute value of the converted string.

Requirements

ROUTINE	REQUIRED HEADER
strtoumax	<stdlib.h>
wcstoumax	<stdlib.h> or <wchar.h>
_strtoumax_l	<stdlib.h>
_wcstoumax_l	<stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for strtod.

See also

Data Conversion
Locale
localeconv
setlocale, _wsetlocale
String to Numeric Value Functions
strtod, _strtod_l, wcstod, _wcstod_l
strtoimax, _strtoimax_l, wcstoimax, _wcstoimax_l
strtol, wcstol, _strtol_l, _wcstol_l
strtoul, _strtoul_l, wcstoul, _wcstoul_l
strtoll, _strtoll_l, wcstoll, _wcstoll_l
atof, _atof_l, _wtof, _wtof_l

strupr, wcsupr

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant _strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr or security-enhanced _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l instead.

_strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr

10/31/2018 • 2 minutes to read • Edit Online

Converts a string to uppercase. More secure versions of these functions are available; see _strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l.

> **IMPORTANT**
>
> **_mbsupr** and **_mbsupr_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
char *_strupr(
   char *str
);
wchar_t *_wcsupr(
   wchar_t *str
);
unsigned char *_mbsupr(
   unsigned char *str
);
char *_strupr_l(
   char *str,
   _locale_t locale
);
wchar_t *_wcsupr_l(
   wchar_t *str,
   _locale_t locale
);
unsigned char *_mbsupr_l(
   unsigned char *str,
   _locale_t locale
);
template <size_t size>
char *_strupr(
   char (&str)[size]
); // C++ only
template <size_t size>
wchar_t *_wcsupr(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
unsigned char *_mbsupr(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
char *_strupr_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
wchar_t *_wcsupr_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
unsigned char *_mbsupr_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str
String to capitalize.

locale
The locale to use.

Return Value

Returns a pointer to the altered string. Because the modification is done in place, the pointer returned is the same as the pointer passed as the input argument. No return value is reserved to indicate an error.

Remarks

The **_strupr** function converts, in place, each lowercase letter in *str* to uppercase. The conversion is determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale; the versions with the **_l** suffix are identical except that they use the locale passed in instead. For more information, see Locale.

_wcsupr and **_mbsupr** are wide-character and multibyte-character versions of **_strupr**. The argument and return value of **_wcsupr** are wide-character strings; those of **_mbsupr** are multibyte-character strings. These three functions behave identically otherwise.

If *str* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions return the original string and set **errno** to **EINVAL**.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsupr	**_strupr**	**_mbsupr**	**_wcsupr**
_tcsupr_l	**_strupr_l**	**_mbsupr_l**	**_wcsupr_l**

Requirements

ROUTINE	REQUIRED HEADER
_strupr, **_strupr_l**	<string.h>
_wcsupr, **_wcsupr_l**	<string.h> or <wchar.h>
_mbsupr, **_mbsupr_l**	<mbstring.h>

For additional compatibility information, see Compatibility.

Example

See the example for _strlwr.

See also

Locale
String Manipulation
_strlwr, _wcslwr, _mbslwr, _strlwr_l, _wcslwr_l, _mbslwr_l

_strupr_s, _strupr_s_l, _mbsupr_s, _mbsupr_s_l, _wcsupr_s, _wcsupr_s_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a string to uppercase, by using the current locale or a specified locale that's passed in. These versions of _strupr, _strupr_l, _mbsupr, _mbsupr_l, _wcsupr_l, _wcsupr have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
>
> **_mbsupr_s** and **_mbsupr_s_l** cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
errno_t _strupr_s(
   char *str,
   size_t numberOfElements
);
errno_t _wcsupr_s(
   wchar_t * str,
   size_t numberOfElements
);
errno_t _strupr_s_l(
   char * str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _wcsupr_s_l(
   wchar_t * str,
   size_t numberOfElements,
   _locale_t locale
);
errno_t _mbsupr_s(
   unsigned char *str,
   size_t numberOfElements
);
errno_t _mbsupr_s_l(
   unsigned char *str,
   size_t numberOfElements,
   _locale_t locale
);
template <size_t size>
errno_t _strupr_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _wcsupr_s(
   wchar_t (&str)[size]
); // C++ only
template <size_t size>
errno_t _strupr_s_l(
   char (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _wcsupr_s_l(
   wchar_t (&str)[size],
   _locale_t locale
); // C++ only
template <size_t size>
errno_t _mbsupr_s(
   unsigned char (&str)[size]
); // C++ only
template <size_t size>
errno_t _mbsupr_s_l(
   unsigned char (&str)[size],
   _locale_t locale
); // C++ only
```

Parameters

str
String to capitalize.

numberOfElements
Size of the buffer.

locale
The locale to use.

Return Value

Zero if successful; a non-zero error code on failure.

These functions validate their parameters. If *str* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, the functions return **EINVAL** and set **errno** to **EINVAL**. If *numberOfElements* is less than the length of the string, the functions return **ERANGE** and set **errno** to **ERANGE**.

Remarks

The **_strupr_s** function converts, in place, each lowercase letter in *str* to uppercase. **_wcsupr_s** is the wide-character version of **_strupr_s**. **_mbsupr_s** is the multi-byte character version of **_strupr_s**.

The conversion is determined by the **LC_CTYPE** category setting of the locale. Other characters are not affected. For more information on **LC_CTYPE**, see setlocale. The versions of these functions without the **_l** suffix use the current locale; the visions with the **_l** suffix are identical except that they use the locale passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

The debug versions of these functions first fill the buffer with 0xFD. To disable this behavior, use _CrtSetDebugFillThreshold.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsupr_s	**_strupr_s**	**_mbsupr_s**	**_wcsupr_s**
_tcsupr_s_l	**_strupr_s_l**	**_mbsupr_s_l**	**_wcsupr_s_l**

Requirements

ROUTINE	REQUIRED HEADER
_strupr_s, **_strupr_s_l**	<string.h>
_wcsupr_s, **_wcsupr_s_l**, **_mbsupr_s**, **_mbsupr_s_l**	<string.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for _strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l .

See also

Locale
Interpretation of Multibyte-Character Sequences
String Manipulation
_strlwr_s, _strlwr_s_l, _mbslwr_s, _mbslwr_s_l, _wcslwr_s, _wcslwr_s_l

strxfrm, wcsxfrm, _strxfrm_l, _wcsxfrm_l

10/31/2018 • 3 minutes to read • Edit Online

Transform a string based on locale-specific information.

Syntax

```
size_t strxfrm(
   char *strDest,
   const char *strSource,
   size_t count
);
size_t wcsxfrm(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count
);
size_t _strxfrm_l(
   char *strDest,
   const char *strSource,
   size_t count,
   _locale_t locale
);
size_t wcsxfrm_l(
   wchar_t *strDest,
   const wchar_t *strSource,
   size_t count,
   _locale_t locale
);
```

Parameters

strDest
Destination string.

strSource
Source string.

count
Maximum number of characters to place in *strDest*.

locale
The locale to use.

Return Value

Returns the length of the transformed string, not counting the terminating null character. If the return value is greater than or equal to *count*, the content of *strDest* is unpredictable. On an error, each function sets **errno** and returns **INT_MAX**. For an invalid character, **errno** is set to **EILSEQ**.

Remarks

The **strxfrm** function transforms the string pointed to by *strSource* into a new collated form that is stored in *strDest*. No more than *count* characters, including the null character, are transformed and placed into the resulting string. The transformation is made using the locale's **LC_COLLATE** category setting. For more

information on **LC_COLLATE**, see setlocale. **strxfrm** uses the current locale for its locale-dependent behavior; **_strxfrm_l** is identical except that it uses the locale passed in instead of the current locale. For more information, see Locale.

After the transformation, a call to **strcmp** with the two transformed strings yields results identical to those of a call to **strcoll** applied to the original two strings. As with **strcoll** and **stricoll**, **strxfrm** automatically handles multibyte-character strings as appropriate.

wcsxfrm is a wide-character version of **strxfrm**; the string arguments of **wcsxfrm** are wide-character pointers. For **wcsxfrm**, after the string transformation, a call to **wcscmp** with the two transformed strings yields results identical to those of a call to **wcscoll** applied to the original two strings. **wcsxfrm** and **strxfrm** behave identically otherwise. **wcsxfrm** uses the current locale for its locale-dependent behavior; **_wcsxfrm_l** uses the locale passed in instead of the current locale.

These functions validate their parameters. If *strSource* is a null pointer, or *strDest* is a **NULL** pointer (unless count is zero), or if *count* is greater than **INT_MAX**, the invalid parameter handler is invoked, as described in Parameter Validation . If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **INT_MAX**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tcsxfrm	**strxfrm**	**strxfrm**	**wcsxfrm**
_tcsxfrm_l	**_strxfrm_l**	**_strxfrm_l**	**_wcsxfrm_l**

In the "C" locale, the order of the characters in the character set (ASCII character set) is the same as the lexicographic order of the characters. However, in other locales, the order of characters in the character set may differ from the lexicographic character order. For example, in certain European locales, the character 'a' (value 0x61) precedes the character 'ä' (value 0xE4) in the character set, but the character 'ä' precedes the character 'a' lexicographically.

In locales for which the character set and the lexicographic character order differ, use **strxfrm** on the original strings and then **strcmp** on the resulting strings to produce a lexicographic string comparison according to the current locale's **LC_COLLATE** category setting. Thus, to compare two strings lexicographically in the above locale, use **strxfrm** on the original strings, then **strcmp** on the resulting strings. Alternately, you can use **strcoll** rather than **strcmp** on the original strings.

strxfrm is basically a wrapper around LCMapString with **LCMAP_SORTKEY**.

The value of the following expression is the size of the array needed to hold the **strxfrm** transformation of the source string:

```
1 + strxfrm( NULL, string, 0 )
```

In the "C" locale only, **strxfrm** is equivalent to the following:

```
strncpy( _string1, _string2, _count );
return( strlen( _string1 ) );
```

Requirements

ROUTINE	REQUIRED HEADER
strxfrm	<string.h>
wcsxfrm	<string.h> or <wchar.h>
_strxfrm_l	<string.h>
_wcsxfrm_l	<string.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
localeconv
setlocale, _wsetlocale
Locale
String Manipulation
strcoll Functions
strcmp, wcscmp, _mbscmp
strncmp, wcsncmp, _mbsncmp, _mbsncmp_l

swab

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _swab instead.

_swab

10/31/2018 • 2 minutes to read • Edit Online

Swaps bytes.

Syntax

```
void _swab(
   char *src,
   char *dest,
   int n
);
```

Parameters

src
Data to be copied and swapped.

dest
Storage location for swapped data.

n
Number of bytes to be copied and swapped.

Return value

The **swab** function does not return a value. The function sets **errno** to **EINVAL** if either the *src* or *dest* pointer is null or *n* is less than zero, and the invalid parameter handler is invoked, as described in Parameter Validation.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this and other return codes.

Remarks

If *n* is even, the **_swab** function copies *n* bytes from *src*, swaps each pair of adjacent bytes, and stores the result at *dest*. If *n* is odd, **_swab** copies and swaps the first *n*-1 bytes of *src*, and the final byte is not copied. The **_swab** function is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Requirements

ROUTINE	REQUIRED HEADER
_swab	C: <stdlib.h> C++: <cstdlib> or <stdlib.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_swab.c

#include <stdlib.h>
#include <stdio.h>

char from[] = "BADCFEHGJILKNMPORQTSVUXWZY";
char to[] =   ".........................";

int main()
{
    printf("Before: %s  %d bytes\n        %s\n\n", from, sizeof(from), to);
    _swab(from, to, sizeof(from));
    printf("After:  %s\n        %s\n\n", from, to);
}
```

```
Before: BADCFEHGJILKNMPORQTSVUXWZY  27 bytes
        .........................

After:  BADCFEHGJILKNMPORQTSVUXWZY
        ABCDEFGHIJKLMNOPQRSTUVWXYZ.
```

See also

Buffer Manipulation

system, _wsystem

11/8/2018 • 2 minutes to read • Edit Online

Executes a command.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int system(
   const char *command
);
int _wsystem(
   const wchar_t *command
);
```

Parameters

command
The command to be executed.

Return Value

If *command* is **NULL** and the command interpreter is found, returns a nonzero value. If the command interpreter is not found, returns 0 and sets **errno** to **ENOENT**. If *command* is not **NULL**, **system** returns the value that is returned by the command interpreter. It returns the value 0 only if the command interpreter returns the value 0. A return value of - 1 indicates an error, and **errno** is set to one of the following values:

E2BIG	The argument list (which is system-dependent) is too big.
ENOENT	The command interpreter cannot be found.
ENOEXEC	The command-interpreter file cannot be executed because the format is not valid.
ENOMEM	Not enough memory is available to execute command; or available memory has been corrupted; or a non-valid block exists, which indicates that the process that's making the call was not allocated correctly.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information about these return codes.

Remarks

The **system** function passes *command* to the command interpreter, which executes the string as an

operating-system command. **system** uses the **COMSPEC** and **PATH** environment variables to locate the command-interpreter file CMD.exe. If *command* is **NULL**, the function just checks whether the command interpreter exists.

You must explicitly flush, by using fflush or _flushall, or close any stream before you call **system**.

_wsystem is a wide-character version of **system**; the *command* argument to **_wsystem** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tsystem	**system**	**system**	**_wsystem**

Requirements

ROUTINE	REQUIRED HEADER
system	<process.h> or <stdlib.h>
_wsystem	<process.h> or <stdlib.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

This example uses **system** to TYPE a text file.

```
// crt_system.c

#include <process.h>

int main( void )
{
   system( "type crt_system.txt" );
}
```

Input: crt_system.txt

```
Line one.
Line two.
```

Output

```
Line one.
Line two.
```

See also

Process and Environment Control
_exec, _wexec Functions
exit, _Exit, _exit
_flushall

tan, tanf, tanl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the tangent.

Syntax

```
double tan( double x );
float tanf( float x );
long double tanl( long double x );
```

```
float tan( float x );  // C++ only
long double tan( long double x );  // C++ only
```

Parameters

x
Angle in radians.

Return value

The **tan** functions return the tangent of *x*. If *x* is greater than or equal to 263, or less than or equal to -263, a loss of significance in the result occurs.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	none	_DOMAIN
± INF	**INVALID**	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **tan** that take and return **float** or **long double** values. In a C program, **tan** always takes and returns **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
tan, **tanf**, **tanl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_tan.c
// This program displays the tangent of pi / 4
// Compile by using: cl crt_tan.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x;

   x = tan( pi / 4 );
   printf( "tan( %f ) = %f\n", pi/4, x );
}
```

```
tan( 0.785398 ) = 1.000000
```

See also

Floating-Point Support
acos, acosf, acosl
asin, asinf, asinl
atan, atanf, atanl, atan2, atan2f, atan2l
cos, cosf, cosl
sin, sinf, sinl
_CItan

tanh, tanhf, tanhl

10/31/2018 • 2 minutes to read • Edit Online

Calculates the hyperbolic tangent.

Syntax

```
double tanh( double x );
float tanhf( float x );
long double tanhl( long double x );
```

```
float tanh( float x );  // C++ only
long double tanh( long double x );  // C++ only
```

Parameters

x
Angle in radians.

Return value

The **tanh** functions return the hyperbolic tangent of *x*. There is no error return.

INPUT	SEH EXCEPTION	MATHERR EXCEPTION
± QNAN,IND	none	_DOMAIN

Remarks

Because C++ allows overloading, you can call overloads of **tanh** that take and return **float** or **long double** values. In a C program, **tanh** always takes and returns **double**.

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C)
tanh, **tanhf**, **tanhl**	<math.h>	<cmath> or <math.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_tanh.c
// This program displays the tangent of pi / 4
// and the hyperbolic tangent of the result.
// Compile by using: cl crt_tanh.c

#include <math.h>
#include <stdio.h>

int main( void )
{
   double pi = 3.1415926535;
   double x, y;

   x = tan( pi / 4 );
   y = tanh( x );
   printf( "tan( %f ) = %f\n", pi/4, x );
   printf( "tanh( %f ) = %f\n", x, y );
}
```

```
tan( 0.785398 ) = 1.000000
tanh( 1.000000 ) = 0.761594
```

See also

Floating-Point Support
acosh, acoshf, acoshl
asinh, asinhf, asinhl
atanh, atanhf, atanhl
cosh, coshf, coshl
sinh, sinhf, sinhl

tell

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _tell instead.

_tell, _telli64

10/31/2018 • 2 minutes to read • Edit Online

Get the position of the file pointer.

Syntax

```
long _tell(
   int handle
);
__int64 _telli64(
   int handle
);
```

Parameters

handle
File descriptor referring to open file.

Return Value

The current position of the file pointer. On devices incapable of seeking, the return value is undefined.

A return value of -1L indicates an error. If *handle* is an invalid file descriptor, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EBADF** and return -1L.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on this, and other, return codes.

Remarks

The **_tell** function gets the current position of the file pointer (if any) associated with the *handle* argument. The position is expressed as the number of bytes from the beginning of the file. For the **_telli64** function, this value is expressed as a 64-bit integer.

Requirements

ROUTINE	REQUIRED HEADER
_tell, **_telli64**	<io.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_tell.c
// This program uses _tell to tell the
// file pointer position after a file read.
//

#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <share.h>
#include <string.h>

int main( void )
{
   int  fh;
   char buffer[500];

   if ( _sopen_s( &fh, "crt_tell.txt", _O_RDONLY, _SH_DENYNO, 0) )
   {
      char buff[50];
      _strerror_s( buff, sizeof(buff), NULL );
      printf( buff );
      exit( -1 );
   }

   if( _read( fh, buffer, 500 ) > 0 )
      printf( "Current file position is: %d\n", _tell( fh ) );
   _close( fh );
}
```

Input: crt_tell.txt

```
Line one.
Line two.
```

Output

```
Current file position is: 20
```

See also

Low-Level I/O
ftell, _ftelli64
_lseek, _lseeki64

tempnam

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _tempnam instead.

_tempnam, _wtempnam, tmpnam, _wtmpnam

10/31/2018 • 4 minutes to read • Edit Online

Generate names you can use to create temporary files. More secure versions of some of these functions are available; see tmpnam_s, _wtmpnam_s.

Syntax

```
char *_tempnam(
   const char *dir,
   const char *prefix
);
wchar_t *_wtempnam(
   const wchar_t *dir,
   const wchar_t *prefix
);
char *tmpnam(
   char *str
);
wchar_t *_wtmpnam(
   wchar_t *str
);
```

Parameters

prefix
The string that will be pre-pended to names returned by **_tempnam**.

dir
The path used in the file name if there is no TMP environment variable, or if TMP is not a valid directory.

str
Pointer that will hold the generated name and will be identical to the name returned by the function. This is a convenient way to save the generated name.

Return Value

Each of these functions returns a pointer to the name generated or **NULL** if there is a failure. Failure can occur if you attempt more than **TMP_MAX** (see STDIO.H) calls with **tmpnam** or if you use **_tempnam** and there is an invalid directory name specified in the TMP environment variable and in the *dir* parameter.

> **NOTE**
> The pointers returned by **tmpnam** and **_wtmpnam** point to internal static buffers. free should not be called to deallocate those pointers. **free** needs to be called for pointers allocated by **_tempnam** and **_wtempnam**.

Remarks

Each of these functions returns the name of a file that does not currently exist. **tmpnam** returns a name that's unique in the designated Windows temporary directory returned by GetTempPathW. **_tempnam** generates a unique name in a directory other than the designated one. Note than when a file name is pre-pended with a backslash and no path information, such as \fname21, this indicates that the name is valid for the current

working directory.

For **tmpnam**, you can store this generated file name in *str*. If *str* is **NULL**, then **tmpnam** leaves the result in an internal static buffer. Thus any subsequent calls destroy this value. The name generated by **tmpnam** consists of a program-generated file name and, after the first call to **tmpnam**, a file extension of sequential numbers in base 32 (.1-.vvu, when **TMP_MAX** in STDIO.H is 32,767).

_tempnam will generate a unique file name for a directory chosen by the following rules:

- If the TMP environment variable is defined and set to a valid directory name, unique file names will be generated for the directory specified by TMP.
- If the TMP environment variable is not defined or if it is set to the name of a directory that does not exist, **_tempnam** will use the *dir* parameter as the path for which it will generate unique names.
- If the TMP environment variable is not defined or if it is set to the name of a directory that does not exist, and if *dir* is either **NULL** or set to the name of a directory that does not exist, **_tempnam** will use the current working directory to generate unique names. Currently, if both TMP and *dir* specify names of directories that do not exist, the **_tempnam** function call will fail.

The name returned by **_tempnam** will be a concatenation of *prefix* and a sequential number, which will combine to create a unique file name for the specified directory. **_tempnam** generates file names that have no extension. **_tempnam** uses malloc to allocate space for the filename; the program is responsible for freeing this space when it is no longer needed.

_tempnam and **tmpnam** automatically handle multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the OEM code page obtained from the operating system. **_wtempnam** is a wide-character version of **_tempnam**; the arguments and return value of **_wtempnam** are wide-character strings. **_wtempnam** and **_tempnam** behave identically except that **_wtempnam** does not handle multibyte-character strings. **_wtmpnam** is a wide-character version of **tmpnam**; the argument and return value of **_wtmpnam** are wide-character strings. **_wtmpnam** and **tmpnam** behave identically except that **_wtmpnam** does not handle multibyte-character strings.

If **_DEBUG** and **_CRTDBG_MAP_ALLOC** are defined, **_tempnam** and **_wtempnam** are replaced by calls to _tempnam_dbg and _wtempnam_dbg.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ttmpnam	**tmpnam**	**tmpnam**	**_wtmpnam**
_ttempnam	**_tempnam**	**_tempnam**	**_wtempnam**

Requirements

ROUTINE	REQUIRED HEADER
_tempnam	<stdio.h>
_wtempnam, **_wtmpnam**	<stdio.h> or <wchar.h>
tmpnam	<stdio.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_tempnam.c
// compile with: /W3
// This program uses tmpnam to create a unique filename in the
// temporary directory, and _tempname to create a unique filename
// in C:\\tmp.

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char * name1 = NULL;
   char * name2 = NULL;
   char * name3 = NULL;

   // Create a temporary filename for the current working directory:
   if ((name1 = tmpnam(NULL)) != NULL) { // C4996
   // Note: tmpnam is deprecated; consider using tmpnam_s instead
      printf("%s is safe to use as a temporary file.\n", name1);
   } else {
      printf("Cannot create a unique filename\n");
   }

   // Create a temporary filename in temporary directory with the
   // prefix "stq". The actual destination directory may vary
   // depending on the state of the TMP environment variable and
   // the global variable P_tmpdir.

   if ((name2 = _tempnam("c:\\tmp", "stq")) != NULL) {
      printf("%s is safe to use as a temporary file.\n", name2);
   } else {
      printf("Cannot create a unique filename\n");
   }

   // When name2 is no longer needed:
   if (name2) {
      free(name2);
   }

   // Unset TMP environment variable, then create a temporary filename in C:\tmp.
   if (_putenv("TMP=") != 0) {
      printf("Could not remove TMP environment variable.\n");
   }

   // With TMP unset, we will use C:\tmp as the temporary directory.
   // Create a temporary filename in C:\tmp with prefix "stq".
   if ((name3 = _tempnam("c:\\tmp", "stq")) != NULL) {
      printf("%s is safe to use as a temporary file.\n", name3);
   }
   else {
      printf("Cannot create a unique filename\n");
   }

   // When name3 is no longer needed:
   if (name3) {
      free(name3);
   }

   return 0;
}
```

```
C:\Users\LocalUser\AppData\Local\Temp\sriw.0 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\stq2 is safe to use as a temporary file.
c:\tmp\stq3 is safe to use as a temporary file.
```

See also

Stream I/O
_getmbcp
malloc
_setmbcp
tmpfile
tmpfile_s

_tempnam_dbg, _wtempnam_dbg

10/31/2018 • 2 minutes to read • Edit Online

Function versions of _tempnam, _wtempnam, tmpnam, _wtmpnam that use the debug version of **malloc**, **_malloc_dbg**.

Syntax

```
char *_tempnam_dbg(
   const char *dir,
   const char *prefix,
   int blockType,
   const char *filename,
   int linenumber
);
wchar_t *_wtempnam_dbg(
   const wchar_t *dir,
   const wchar_t *prefix,
   int blockType,
   const char *filename,
   int linenumber
);
```

Parameters

dir
The path used in the file name if there is no TMP environment variable, or if TMP is not a valid directory.

prefix
The string that will be pre-pended to names returned by **_tempnam**.

blockType
Requested type of memory block: **_CLIENT_BLOCK** or **_NORMAL_BLOCK**.

filename
Pointer to name of source file that requested allocation operation or **NULL**.

linenumber
Line number in source file where allocation operation was requested or **NULL**.

Return Value

Each function returns a pointer to the name generated or **NULL** if there is a failure. Failure can occur if there is an invalid directory name specified in the TMP environment variable and in the *dir* parameter.

> **NOTE**
> **free** (or **free_dbg**) does need to be called for pointers allocated by **_tempnam_dbg** and **_wtempnam_dbg**.

Remarks

The **_tempnam_dbg** and **_wtempnam_dbg** functions are identical to **_tempnam** and **_wtempnam** except that, when **_DEBUG** is defined, these functions use the debug version of **malloc** and **_malloc_dbg**, to allocate

memory if **NULL** is passed as the first parameter. For more information, see _malloc_dbg.

You do not need to call these functions explicitly in most cases. Instead, you can define the flag **_CRTDBG_MAP_ALLOC**. When **_CRTDBG_MAP_ALLOC** is defined, calls to **_tempnam** and **_wtempnam** are remapped to **_tempnam_dbg** and **_wtempnam_dbg**, respectively, with the *blockType* set to **_NORMAL_BLOCK**. Thus, you do not need to call these functions explicitly unless you want to mark the heap blocks as **_CLIENT_BLOCK**. For more information, see Types of blocks on the debug heap.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ttempnam_dbg	**_tempnam_dbg**	**_tempnam_dbg**	**_wtempnam_dbg**

Requirements

ROUTINE	REQUIRED HEADER
_tempnam_dbg, **_wtempnam_dbg**	<crtdbg.h>

For additional compatibility information, see Compatibility.

See also

_tempnam, _wtempnam, tmpnam, _wtmpnam
Stream I/O
Debug Versions of Heap Allocation Functions

terminate (CRT)

10/31/2018 • 2 minutes to read • Edit Online

Calls abort or a function you specify using **set_terminate**.

Syntax

```
void terminate( void );
```

Remarks

The **terminate** function is used with C++ exception handling and is called in the following cases:

- A matching catch handler cannot be found for a thrown C++ exception.
- An exception is thrown by a destructor function during stack unwind.
- The stack is corrupted after throwing an exception.

terminate calls abort by default. You can change this default by writing your own termination function and calling **set_terminate** with the name of your function as its argument. **terminate** calls the last function given as an argument to **set_terminate**. For more information, see Unhandled C++ Exceptions.

Requirements

ROUTINE	REQUIRED HEADER
terminate	<eh.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_terminate.cpp
// compile with: /EHsc
#include <eh.h>
#include <process.h>
#include <iostream>
using namespace std;

void term_func();

int main()
{
    int i = 10, j = 0, result;
    set_terminate( term_func );
    try
    {
        if( j == 0 )
            throw "Divide by zero!";
        else
            result = i/j;
    }
    catch( int )
    {
        cout << "Caught some integer exception.\n";
    }
    cout << "This should never print.\n";
}

void term_func()
{
    cout << "term_func() was called by terminate().\n";

    // ... cleanup tasks performed here

    // If this function does not exit, abort is called.

    exit(-1);
}
```

```
term_func() was called by terminate().
```

See also

Exception Handling Routines
abort
_set_se_translator
set_terminate
set_unexpected
unexpected

tgamma, tgammaf, tgammal

11/9/2018 • 2 minutes to read • Edit Online

Determines the gamma function of the specified value.

Syntax

```
double tgamma(
   double x
);

float tgamma(
   float x
); //C++ only

long double tgamma(
   long double x
); //C++ only

float tgammaf(
   float x
);

long double tgammal(
   long double x
);
```

Parameters

x
The value to find the gamma of.

Return Value

If successful, returns the gamma of *x*.

A range error may occur if the magnitude of *x* is too large or too small for the data type. A domain error or range error may occur if *x* <= 0.

ISSUE	RETURN
x = ±0	±INFINITY
x = negative integer	NaN
x = -INFINITY	NaN
x = +INFINITY	+INFINITY
x = NaN	NaN
domain error	NaN

ISSUE	RETURN
pole error	±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL
overflow range error	±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL
underflow range error	the correct value, after rounding.

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **tgamma** that take and return **float** and **long double** types. In a C program, **tgamma** always takes and returns a **double**.

If x is a natural number, this function returns the factorial of (x-1).

Requirements

FUNCTION	C HEADER	C++ HEADER
tgamma, **tgammaf**, **tgammal**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
lgamma, lgammaf, lgammal

time, _time32, _time64

11/8/2018 • 3 minutes to read • Edit Online

Gets the system time.

Syntax

```
time_t time( time_t *destTime );
__time32_t _time32( __time32_t *destTime );
__time64_t _time64( __time64_t *destTime );
```

Parameters

destTime
Pointer to the storage location for time.

Return Value

Returns the time as seconds elapsed since midnight, January 1, 1970, or -1 in the case of an error.

Remarks

The **time** function returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, Coordinated Universal Time (UTC), according to the system clock. The return value is stored in the location given by *destTime*. This parameter may be **NULL**, in which case the return value is not stored.

time is a wrapper for **_time64** and **time_t** is, by default, equivalent to **__time64_t**. If you need to force the compiler to interpret **time_t** as the old 32-bit **time_t**, you can define **_USE_32BIT_TIME_T**. This is not recommended because your application may fail after January 18, 2038; the use of this macro is not allowed on 64-bit platforms.

Requirements

ROUTINE	REQUIRED C HEADER	REQUIRED C++ HEADER
time, **_time32**, **_time64**	<time.h>	<ctime> or <time.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_times.c
// compile with: /W3
// This program demonstrates these time and date functions:
//      time         _ftime    ctime_s     asctime_s
//      _localtime64_s    _gmtime64_s    mktime    _tzset
//      _strtime_s     _strdate_s  strftime
//
// Also the global variable:
//      _tzname
//
// Turn off deprecated unsafe CRT function warnings
```

```
#define _CRT_SECURE_NO_WARNINGS 1

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>

int main()
{
    char tmpbuf[128], timebuf[26], ampm[] = "AM";
    time_t ltime;
    struct _timeb tstruct;
    struct tm today, gmt, xmas = { 0, 0, 12, 25, 11, 93 };
    errno_t err;

    // Set time zone from TZ environment variable. If TZ is not set,
    // the operating system is queried to obtain the default value
    // for the variable.
    //
    _tzset();

    // Display operating system-style date and time.
    _strtime_s( tmpbuf, 128 );
    printf( "OS time:\t\t\t\t%s\n", tmpbuf );
    _strdate_s( tmpbuf, 128 );
    printf( "OS date:\t\t\t\t%s\n", tmpbuf );

    // Get UNIX-style time and display as number and string.
    time( &ltime );
    printf( "Time in seconds since UTC 1/1/70:\t%lld\n", (long long)ltime );
    err = ctime_s(timebuf, 26, &ltime);
    if (err)
    {
        printf("ctime_s failed due to an invalid argument.");
        exit(1);
    }
    printf( "UNIX time and date:\t\t\t%s", timebuf );

    // Display UTC.
    err = _gmtime64_s( &gmt, &ltime );
    if (err)
    {
        printf("_gmtime64_s failed due to an invalid argument.");
    }
    err = asctime_s(timebuf, 26, &gmt);
    if (err)
    {
        printf("asctime_s failed due to an invalid argument.");
        exit(1);
    }
    printf( "Coordinated universal time:\t\t%s", timebuf );

    // Convert to time structure and adjust for PM if necessary.
    err = _localtime64_s( &today, &ltime );
    if (err)
    {
        printf("_localtime64_s failed due to an invalid argument.");
        exit(1);
    }
    if ( today.tm_hour >= 12 )
    {
        strcpy_s( ampm, sizeof(ampm), "PM" );
        today.tm_hour -= 12;
    }
    if ( today.tm_hour == 0 )  // Adjust if midnight hour.
        today.tm_hour = 12;
```

```
    // Convert today into an ASCII string
    err = asctime_s(timebuf, 26, &today);
    if (err)
    {
        printf("asctime_s failed due to an invalid argument.");
        exit(1);
    }

    // Note how pointer addition is used to skip the first 11
    // characters and printf is used to trim off terminating
    // characters.
    //
    printf( "12-hour time:\t\t\t\t%.8s %s\n",
        timebuf + 11, ampm );

    // Print additional time information.
    _ftime( &tstruct ); // C4996
    // Note: _ftime is deprecated; consider using _ftime_s instead
    printf( "Plus milliseconds:\t\t\t%u\n", tstruct.millitm );
    printf( "Zone difference in hours from UTC:\t%u\n",
             tstruct.timezone/60 );
    printf( "Time zone name:\t\t\t\t%s\n", _tzname[0] ); //C4996
    // Note: _tzname is deprecated; consider using _get_tzname
    printf( "Daylight savings:\t\t\t%s\n",
             tstruct.dstflag ? "YES" : "NO" );

    // Make time for noon on Christmas, 1993.
    if( mktime( &xmas ) != (time_t)-1 )
    {
        err = asctime_s(timebuf, 26, &xmas);
        if (err)
        {
            printf("asctime_s failed due to an invalid argument.");
            exit(1);
        }
        printf( "Christmas\t\t\t\t%s\n", timebuf );
    }

    // Use time structure to build a customized time string.
    err = _localtime64_s( &today, &ltime );
    if (err)
    {
        printf(" _localtime64_s failed due to invalid arguments.");
        exit(1);
    }

    // Use strftime to build a customized time string.
    strftime( tmpbuf, 128,
              "Today is %A, day %d of %B in the year %Y.\n", &today );
    printf( tmpbuf );
}
```

```
OS time:            13:51:23
OS date:            04/25/03
Time in seconds since UTC 1/1/70:   1051303883
UNIX time and date:         Fri Apr 25 13:51:23 2003
Coordinated universal time:      Fri Apr 25 20:51:23 2003
12-hour time:            01:51:23 PM
Plus milliseconds:          552
Zone difference in hours from UTC:   8
Time zone name:             Pacific Standard Time
Daylight savings:          YES
Christmas            Sat Dec 25 12:00:00 1993

Today is Friday, day 25 of April in the year 2003.
```

See also

Time Management
asctime, _wasctime
asctime_s, _wasctime_s
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
localtime_s, _localtime32_s, _localtime64_s
_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64

timespec_get, _timespec32_get, _timespec64_get

11/9/2018 • 2 minutes to read • Edit Online

Sets the interval pointed to by the first argument to the current calendar time, based on the specified time base.

Syntax

```
int timespec_get(
    struct timespec* const time_spec,
    int const base
);
int _timespec32_get(
    struct _timespec32* const time_spec,
    int const base
);
int _timespec64_get(
    struct _timespec64* const time_spec,
    int const base
);
```

Parameters

time_spec
Pointer to a struct that is set to the time in seconds and nanoseconds since the start of the epoch.

base
A non-zero implementation-specific value that specifies the time base.

Return Value

The value of *base* if successful, otherwise it returns zero.

Remarks

The **timespec_get** functions set the current time in the struct pointed to by the *time_spec* argument. All versions of this struct have two members, **tv_sec** and **tv_nsec**. The **tv_sec** value is set to the whole number of seconds and **tv_nsec** to the integral number of nanoseconds, rounded to the resolution of the system clock, since the start of the epoch specified by *base*.

Microsoft Specific

These functions support only **TIME_UTC** as the *base* value. This sets the *time_spec* value to the number of seconds and nanoseconds since the epoch start, Midnight, January 1, 1970, Coordinated Universal Time (UTC). In a **struct _timespec32**, **tv_sec** is a **__time32_t** value. In a **struct _timespec64**, **tv_sec** is a **__time64_t** value. In a **struct timespec**, **tv_sec** is a **time_t** type, which is 32 bits or 64 bits in length depending on whether the preprocessor macro _USE_32BIT_TIME_T is defined. The **timespec_get** function is an inline function that calls **_timespec32_get** if _USE_32BIT_TIME_T is defined; otherwise it calls **_timespec64_get**.

End Microsoft Specific

Requirements

ROUTINE	REQUIRED HEADER
timespec_get, **_timespec32_get**, **_timespec64_get**	C: <time.h>, C++: <ctime> or <time.h>

For additional compatibility information, see Compatibility.

See also

Time Management
asctime, _wasctime
asctime_s, _wasctime_s
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
gmtime_s, _gmtime32_s, _gmtime64_s
localtime, _localtime32, _localtime64
localtime_s, _localtime32_s, _localtime64_s
time, _time32, _time64
_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64

tmpfile

10/31/2018 • 2 minutes to read • Edit Online

Creates a temporary file. This function is deprecated because a more secure version is available; see tmpfile_s.

Syntax

```
FILE *tmpfile( void );
```

Return Value

If successful, **tmpfile** returns a stream pointer. Otherwise, it returns a **NULL** pointer.

Remarks

The **tmpfile** function creates a temporary file and returns a pointer to that stream. The temporary file is created in the root directory. To create a temporary file in a directory other than the root, use tmpnam or tempnam in conjunction with fopen.

If the file cannot be opened, **tmpfile** returns a **NULL** pointer. This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **_rmtmp** is called, assuming that the current working directory does not change. The temporary file is opened in **w+b** (binary read/write) mode.

Failure can occur if you attempt more than TMP_MAX (see STDIO.H) calls with **tmpfile**.

Requirements

ROUTINE	REQUIRED HEADER
tmpfile	<stdio.h>

For additional compatibility information, see Compatibility.

Example

> **NOTE**
> This example requires administrative privileges to run on Windows Vista.

```
// crt_tmpfile.c
// compile with: /W3
// This program uses tmpfile to create a
// temporary file, then deletes this file with _rmtmp.
#include <stdio.h>

int main( void )
{
   FILE *stream;
   int  i;

   // Create temporary files.
   for( i = 1; i <= 3; i++ )
   {
      if( (stream = tmpfile()) == NULL ) // C4996
      // Note: tmpfile is deprecated; consider using tmpfile_s instead
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }

   // Remove temporary files.
   printf( "%d temporary files deleted\n", _rmtmp() );
}
```

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
3 temporary files deleted
```

See also

Stream I/O

_rmtmp

_tempnam, _wtempnam, tmpnam, _wtmpnam

tmpfile_s

10/31/2018 • 2 minutes to read • Edit Online

Creates a temporary file. It is a version of tmpfile with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t tmpfile_s(
   FILE** pFilePtr
);
```

Parameters

pFilePtr
The address of a pointer to store the address of the generated pointer to a stream.

Return Value

Returns 0 if successful, an error code on failure.

Error Conditions

PFILEPTR	RETURN VALUE	CONTENTS OF *PFILEPTR*
NULL	**EINVAL**	not changed

If the above parameter validation error occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the return value is **EINVAL**.

Remarks

The **tmpfile_s** function creates a temporary file and puts a pointer to that stream in the *pFilePtr* argument. The temporary file is created in the root directory. To create a temporary file in a directory other than the root, use tmpnam_s or tempnam in conjunction with fopen.

If the file cannot be opened, **tmpfile_s** writes **NULL** to the *pFilePtr* parameter. This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **_rmtmp** is called, assuming that the current working directory does not change. The temporary file is opened in **w+b** (binary read/write) mode.

Failure can occur if you attempt more than **TMP_MAX_S** (see STDIO.H) calls with **tmpfile_s**.

Requirements

ROUTINE	REQUIRED HEADER
tmpfile_s	<stdio.h>

For additional compatibility information, see Compatibility.

Example

> **NOTE**
>
> This example may require administrative privileges to run on Windows.

```
// crt_tmpfile_s.c
// This program uses tmpfile_s to create a
// temporary file, then deletes this file with _rmtmp.
//

#include <stdio.h>

int main( void )
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;
   errno_t err;

   // Create temporary files.
   for( i = 1; i <= 3; i++ )
   {
      err = tmpfile_s(&stream);
      if( err )
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }

   // Remove temporary files.
   printf( "%d temporary files deleted\n", _rmtmp() );
}
```

```
Temporary file 1 was created
Temporary file 2 was created
Temporary file 3 was created
3 temporary files deleted
```

See also

Stream I/O

_rmtmp

_tempnam, _wtempnam, tmpnam, _wtmpnam

tmpnam_s, _wtmpnam_s

10/31/2018 • 3 minutes to read • Edit Online

Generate names you can use to create temporary files. These are versions of tmpnam and _wtmpnam with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t tmpnam_s(
   char * str,
   size_t sizeInChars
);
errno_t _wtmpnam_s(
   wchar_t *str,
   size_t sizeInChars
);
template <size_t size>
errno_t tmpnam_s(
   char (&str)[size]
); // C++ only
template <size_t size>
errno_t _wtmpnam_s(
   wchar_t (&str)[size]
); // C++ only
```

Parameters

str
Pointer that will hold the generated name.

sizeInChars
The size of the buffer in characters.

Return Value

Both of these functions return 0 if successful or an error number on failure.

Error Conditions

str	*sizeInChars*	**Return Value**	**Contents of** *str*
NULL	any	**EINVAL**	not modified
not **NULL** (points to valid memory)	too short	**ERANGE**	not modified

If *str* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return **EINVAL**.

Remarks

Each of these functions returns the name of a file that does not currently exist. **tmpnam_s** returns a name unique in the designated Windows temporary directory returned by GetTempPathW. Note than when a file name is pre-

pended with a backslash and no path information, such as \fname21, this indicates that the name is valid for the current working directory.

For **tmpnam_s**, you can store this generated file name in *str*. The maximum length of a string returned by **tmpnam_s** is **L_tmpnam_s**, defined in STDIO.H. If *str* is **NULL**, then **tmpnam_s** leaves the result in an internal static buffer. Thus any subsequent calls destroy this value. The name generated by **tmpnam_s** consists of a program-generated file name and, after the first call to **tmpnam_s**, a file extension of sequential numbers in base 32 (.1-.1vvvvvu, when **TMP_MAX_S** in STDIO.H is **INT_MAX**).

tmpnam_s automatically handles multibyte-character string arguments as appropriate, recognizing multibyte-character sequences according to the OEM code page obtained from the operating system. **_wtmpnam_s** is a wide-character version of **tmpnam_s**; the argument and return value of **_wtmpnam_s** are wide-character strings. **_wtmpnam_s** and **tmpnam_s** behave identically except that **_wtmpnam_s** does not handle multibyte-character strings.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically, eliminating the need to specify a size argument. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ttmpnam_s	**tmpnam_s**	**tmpnam_s**	**_wtmpnam_s**

Requirements

ROUTINE	REQUIRED HEADER
tmpnam_s	<stdio.h>
_wtmpnam_s	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_tmpnam_s.c
// This program uses tmpnam_s to create a unique filename in the
// current working directory.
//

#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   char name1[L_tmpnam_s];
   errno_t err;
   int i;

   for (i = 0; i < 15; i++)
   {
      err = tmpnam_s( name1, L_tmpnam_s );
      if (err)
      {
         printf("Error occurred creating unique filename.\n");
         exit(1);
      }
      else
      {
         printf( "%s is safe to use as a temporary file.\n", name1 );
      }
   }
}
```

```
C:\Users\LocalUser\AppData\Local\Temp\u19q8.0 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.1 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.2 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.3 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.4 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.5 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.6 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.7 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.8 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.9 is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.a is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.b is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.c is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.d is safe to use as a temporary file.
C:\Users\LocalUser\AppData\Local\Temp\u19q8.e is safe to use as a temporary file.
```

See also

Stream I/O
_getmbcp
malloc
_setmbcp
tmpfile_s

toascii, __toascii

10/31/2018 • 2 minutes to read • Edit Online

Converts characters to 7-bit ASCII by truncation.

Syntax

```
int __toascii(
   int c
);
#define toascii __toascii
```

Parameters

c
Character to convert.

Return Value

__toascii converts the value of *c* to the 7-bit ASCII range and returns the result. There is no return value reserved to indicate an error.

Remarks

The **__toascii** routine converts the given character to an ASCII character by truncating it to the low-order 7 bits. No other transformation is applied.

The **__toascii** routine is defined as a macro unless the preprocessor macro _CTYPE_DISABLE_MACROS is defined. For backward compatibility, **toascii** is defined as a macro only when __STDC__ is not defined or is defined as 0; otherwise it is undefined.

Requirements

ROUTINE	REQUIRED HEADER
toascii, **__toascii**	C: <ctype.h> C++: <cctype> or <ctype.h>

The **toascii** macro is a POSIX extension, and **__toascii** is a Microsoft-specific implementation of the POSIX extension. For additional compatibility information, see Compatibility.

See also

Data Conversion
is, isw Routines
to Functions

tolower, _tolower, towlower, _tolower_l, _towlower_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a character to lowercase.

Syntax

```
int tolower(
   int c
);
int _tolower(
   int c
);
int towlower(
   wint_t c
);
int _tolower_l(
   int c,
   _locale_t locale
);
int _towlower_l(
   wint_t c,
   _locale_t locale
);
```

Parameters

c
Character to convert.

locale
Locale to use for locale-specific translation.

Return Value

Each of these routines converts a copy of *c* to lower case if the conversion is possible, and returns the result. There is no return value reserved to indicate an error.

Remarks

Each of these routines converts a given uppercase letter to a lowercase letter if it is possible and relevant. The case conversion of **towlower** is locale-specific. Only the characters relevant to the current locale are changed in case. The functions without the **_l** suffix use the currently set locale. The versions of these functions that have the **_l** suffix take the locale as a parameter and use that instead of the currently set locale. For more information, see Locale.

In order for **_tolower** to give the expected results, __isascii and isupper must both return nonzero.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_totlower	**tolower**	**_mbctolower**	**towlower**

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_totlower_l	**_tolower_l**	**_mbctolower_l**	**_towlower_l**

> **NOTE**
>
> **_tolower_l** and **_towlower_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_totlower_l**.

Requirements

ROUTINE	REQUIRED HEADER
tolower	<ctype.h>
_tolower	<ctype.h>
towlower	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example in to Functions.

See also

Data Conversion
is, isw Routines
to Functions
Locale
Interpretation of Multibyte-Character Sequences

toupper, _toupper, towupper, _toupper_l, _towupper_l

3/1/2019 • 2 minutes to read • Edit Online

Convert character to uppercase.

Syntax

```
int toupper(
   int c
);
int _toupper(
   int c
);
int towupper(
   wint_t c
);
int _toupper_l(
   int c ,
   _locale_t locale
);
int _towupper_l(
   wint_t c ,
   _locale_t locale
);
```

Parameters

c
Character to convert.

locale
Locale to use.

Return Value

Each of these routines converts a copy of *c*, if possible, and returns the result.

If *c* is a wide character for which **iswlower** is nonzero and there is a corresponding wide character for which iswupper is nonzero, **towupper** returns the corresponding wide character; otherwise, **towupper** returns *c* unchanged.

There is no return value reserved to indicate an error.

In order for **toupper** to give the expected results, __isascii and islower must both return nonzero.

Remarks

Each of these routines converts a given lowercase letter to an uppercase letter if possible and appropriate. The case conversion of **towupper** is locale-specific. Only the characters relevant to the current locale are changed in case. The functions without the **_l** suffix use the currently set locale. The versions of these functions with the **_l** suffix take the locale as a parameter and use that instead of the currently set locale. For more information, see Locale.

In order for **toupper** to give the expected results, __isascii and isupper must both return nonzero.

Data Conversion Routines

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_totupper	**toupper**	**_mbctoupper**	**towupper**
_totupper_l	**_toupper_l**	**_mbctoupper_l**	**_towupper_l**

> **NOTE**
> **_toupper_l** and **_towupper_l** have no locale dependence and are not meant to be called directly. They are provided for internal use by **_totupper_l**.

Requirements

ROUTINE	REQUIRED HEADER
toupper	<ctype.h>
_toupper	<ctype.h>
towupper	<ctype.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example in to Functions.

See also

is, isw Routines
to Functions
Locale
Interpretation of Multibyte-Character Sequences

towctrans

10/31/2018 • 2 minutes to read • Edit Online

Transforms a character.

Syntax

```
wint_t towctrans(
   wint_t c,
   wctrans_t category
);
```

Parameters

c
The character you want to transform.

category
An identifier that contains the return value of wctrans.

Return Value

The character *c*, after **towctrans** used the transform rule in *category*.

Remarks

The value of *category* must have been returned by an earlier successful call to wctrans.

Requirements

ROUTINE	REQUIRED HEADER
towctrans	<wctype.h>

For additional compatibility information, see Compatibility.

Example

See **wctrans** for a sample that uses **towctrans**.

See also

Data Conversion

trunc, truncf, truncl

10/31/2018 • 2 minutes to read • Edit Online

Determines the nearest integer that is less than or equal to the specified floating-point value.

Syntax

```
double trunc( double x );
float trunc( float x ); //C++ only
long double truncl( long double x );
```

```
long double trunc( long double x ); //C++ only
float trunc( float x ); //C++ only
```

Parameters

x
The value to truncate.

Return Value

If successful, returns an integer value of *x*, rounded towards zero.

Otherwise, may return one of the following:

ISSUE	RETURN
x = ±INFINITY	x
x = ±0	x
x = NaN	NaN

Errors are reported as specified in _matherr.

Remarks

Because C++ allows overloading, you can call overloads of **trunc** that take and return **float** and **long double** types. In a C program, **trunc** always takes and returns a **double**.

Because the largest floating-point values are exact integers, this function will not overflow on its own. However, you may cause the function to overflow by returning a value into an integer type.

You can also round down by implicitly converting from floating-point to integral; however, doing so is limited to the values that can be stored in the target type.

Requirements

FUNCTION	C HEADER	C++ HEADER
trunc, **truncf**, **truncl**	<math.h>	<cmath>

For additional compatibility information, see Compatibility.

See also

Alphabetical Function Reference
floor, floorf, floorl
ceil, ceilf, ceill
round, roundf, roundl

tzset

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _tzset instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_tzset

11/8/2018 • 3 minutes to read • Edit Online

Sets time environment variables.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
void _tzset( void );
```

Remarks

The **_tzset** function uses the current setting of the environment variable **TZ** to assign values to three global variables: **_daylight**, **_timezone**, and **_tzname**. These variables are used by the _ftime and localtime functions to make corrections from coordinated universal time (UTC) to local time, and by the time function to compute UTC from system time. Use the following syntax to set the **TZ** environment variable:

set TZ=*tzn* [**+**|**-**]*hh*[**:***mm*[**:***ss*]][*dzn*]

PARAMETER	DESCRIPTION
tzn	Three-letter time-zone name, such as PST. You must specify the correct offset from local time to UTC.
hh	Difference in hours between UTC and local time. Sign (+) optional for positive values.
mm	Minutes. Separated from *hh* by a colon (**:**).
ss	Seconds. Separated from *mm* by a colon (**:**).
dzn	Three-letter daylight-saving-time zone such as PDT. If daylight saving time is never in effect in the locality, set **TZ** without a value for *dzn*. The C run-time library assumes the United States' rules for implementing the calculation of daylight saving time (DST).

> **NOTE**
> Take care in computing the sign of the time difference. Because the time difference is the offset from local time to UTC (rather than the reverse), its sign may be the opposite of what you might intuitively expect. For time zones ahead of UTC, the time difference is negative; for those behind UTC, the difference is positive.

For example, to set the **TZ** environment variable to correspond to the current time zone in Germany, enter

the following on the command line:

```
set TZ=GST-1GDT
```

This command uses GST to indicate German standard time, assumes that UTC is one hour behind Germany (or in other words, that Germany is one hour ahead of UTC), and assumes that Germany observes daylight-saving time.

If the **TZ** value is not set, **_tzset** attempts to use the time zone information specified by the operating system. In the Windows operating system, this information is specified in the Date/Time application in Control Panel. If **_tzset** cannot obtain this information, it uses PST8PDT by default, which signifies the Pacific Time zone.

Based on the **TZ** environment variable value, the following values are assigned to the global variables **_daylight**, **_timezone**, and **_tzname** when **_tzset** is called:

GLOBAL VARIABLE	DESCRIPTION	DEFAULT VALUE
_daylight	Nonzero value if a daylight-saving-time zone is specified in **TZ** setting; otherwise, 0.	1
_timezone	Difference in seconds between local time and UTC.	28800 (28800 seconds equals 8 hours)
_tzname[0]	String value of time-zone name from **TZ** environmental variable; empty if **TZ** has not been set.	PST
_tzname[1]	String value of daylight-saving-time zone; empty if daylight-saving-time zone is omitted from **TZ** environmental variable.	PDT

The default values shown in the preceding table for **_daylight** and the **_tzname** array correspond to "PST8PDT." If the DST zone is omitted from the **TZ** environmental variable, the value of **_daylight** is 0 and the _ftime, gmtime, and localtime functions return 0 for their DST flags.

Requirements

ROUTINE	REQUIRED HEADER
_tzset	<time.h>

The **_tzset** function is Microsoft-specific. For more information, see Compatibility.

Example

```
// crt_tzset.cpp
// This program uses _tzset to set the global variables
// named _daylight, _timezone, and _tzname. Since TZ is
// not being explicitly set, it uses the system time.

#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    _tzset();
    int daylight;
    _get_daylight( &daylight );
    printf( "_daylight = %d\n", daylight );
    long timezone;
    _get_timezone( &timezone );
    printf( "_timezone = %ld\n", timezone );
    size_t s;
    char tzname[100];
    _get_tzname( &s, tzname, sizeof(tzname), 0 );
    printf( "_tzname[0] = %s\n", tzname );
    exit( 0 );
}
```

```
_daylight = 1
_timezone = 28800
_tzname[0] = Pacific Standard Time
```

See also

Time Management
asctime, _wasctime
_ftime, _ftime32, _ftime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
time, _time32, _time64
_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64

umask

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _umask or security-enhanced _umask_s instead.

_umask

11/8/2018 • 2 minutes to read • Edit Online

Sets the default file-permission mask. A more secure version of this function is available; see _umask_s.

Syntax

```
int _umask( int pmode );
```

Parameters

pmode
Default permission setting.

Return Value

_umask returns the previous value of *pmode*. There is no error return.

Remarks

The **_umask** function sets the file-permission mask of the current process to the mode specified by *pmode*. The file-permission mask modifies the permission setting of new files created by **_creat**, **_open**, or **_sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\STAT.H:

PMODE	
_S_IWRITE	Writing permitted.
_S_IREAD	Reading permitted.
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). If the *pmode* argument is **_S_IREAD**, reading is not allowed (the file is write-only). If the *pmode* argument is **_S_IWRITE**, writing is not allowed (the file is read-only). For example, if the write bit is set in the mask, any new files will be read-only. Note that with MS-DOS and the Windows operating systems, all files are readable; it is not possible to give write-only permission. Therefore, setting the read bit with **_umask** has no effect on the file's modes.

If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function will simply ignore those.

Requirements

ROUTINE	REQUIRED HEADER

ROUTINE	REQUIRED HEADER
_umask	<io.h>, <sys/stat.h>, <sys/types.h>

For additional compatibility information, see Compatibility.

Libraries

All versions of the C run-time libraries.

Example

```
// crt_umask.c
// compile with: /W3
// This program uses _umask to set
// the file-permission mask so that all future
// files will be created as read-only files.
// It also displays the old mask.
#include <sys/stat.h>
#include <sys/types.h>
#include <io.h>
#include <stdio.h>

int main( void )
{
   int oldmask;

   /* Create read-only files: */
   oldmask = _umask( _S_IWRITE ); // C4996
   // Note: _umask is deprecated; consider using _umask_s instead
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

```
Oldmask = 0x0000
```

See also

File Handling
Low-Level I/O
_chmod, _wchmod
_creat, _wcreat
_mkdir, _wmkdir
_open, _wopen

_umask_s

10/31/2018 • 2 minutes to read • Edit Online

Sets the default file-permission mask. A version of _umask with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t _umask_s(
   int mode,
   int * pOldMode
);
```

Parameters

mode
Default permission setting.

pOldMode
The previous value of the permission setting.

Return Value

Returns an error code if *mode* does not specify a valid mode or the *pOldMode* pointer is **NULL**.

Error Conditions

MODE	*POLDMODE*	RETURN VALUE	CONTENTS OF *POLDMODE*
any	**NULL**	**EINVAL**	not modified
invalid mode	any	**EINVAL**	not modified

If one of the above conditions occurs, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **_umask_s** returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **_umask_s** function sets the file-permission mask of the current process to the mode specified by *mode*. The file-permission mask modifies the permission setting of new files created by **_creat**, **_open**, or **_sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The integer expression *pmode* contains one or both of the following manifest constants, defined in SYS\STAT.H:

PMODE	
_S_IWRITE	Writing permitted.
_S_IREAD	Reading permitted.

PMODE	
_S_IREAD | **_S_IWRITE**	Reading and writing permitted.

When both constants are given, they are joined with the bitwise-OR operator (**|**). If the *mode* argument is **_S_IREAD**, reading is not allowed (the file is write-only). If the *mode* argument is **_S_IWRITE**, writing is not allowed (the file is read-only). For example, if the write bit is set in the mask, any new files will be read-only. Note that with MS-DOS and the Windows operating systems, all files are readable; it is not possible to give write-only permission. Therefore, setting the read bit with **_umask_s** has no effect on the file's modes.

If *pmode* is not a combination of one of the manifest constants or incorporates an alternate set of constants, the function will simply ignore those.

Requirements

ROUTINE	REQUIRED HEADER
_umask_s	<io.h> and <sys/stat.h> and <sys/types.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_umask_s.c
/* This program uses _umask_s to set
* the file-permission mask so that all future
* files will be created as read-only files.
* It also displays the old mask.
*/

#include <sys/stat.h>
#include <sys/types.h>
#include <io.h>
#include <stdio.h>

int main( void )
{
   int oldmask, err;

   /* Create read-only files: */
   err = _umask_s( _S_IWRITE, &oldmask );
   if (err)
   {
      printf("Error setting the umask.\n");
      exit(1);
   }
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

```
Oldmask = 0x0000
```

See also

File Handling
Low-Level I/O
_chmod, _wchmod

_creat, _wcreat
_mkdir, _wmkdir
_open, _wopen
_umask

__uncaught_exception

10/31/2018 • 2 minutes to read • Edit Online

Indicates whether one or more exceptions have been thrown, but have not yet been handled by the corresponding **catch** block of a try-catch statement.

Syntax

```
bool __uncaught_exception(
   );
```

Return Value

true from the time an exception is thrown in a **try** block until the matching **catch** block is initialized; otherwise, **false**.

Remarks

Requirements

ROUTINE	REQUIRED HEADER
__uncaught_exception	eh.h

See also

try, throw, and catch Statements (C++)

unexpected (CRT)

10/31/2018 • 2 minutes to read • Edit Online

Calls **terminate** or function you specify using **set_unexpected**.

Syntax

```
void unexpected( void );
```

Remarks

The **unexpected** routine is not used with the current implementation of C++ exception handling. **unexpected** calls **terminate** by default. You can change this default behavior by writing a custom termination function and calling **set_unexpected** with the name of your function as its argument. **unexpected** calls the last function given as an argument to **set_unexpected**.

Requirements

ROUTINE	REQUIRED HEADER
unexpected	<eh.h>

For additional compatibility information, see Compatibility.

See also

Exception Handling Routines
abort
_set_se_translator
set_terminate
set_unexpected
terminate

ungetc, ungetwc

11/8/2018 • 2 minutes to read • Edit Online

Pushes a character back onto the stream.

Syntax

```
int ungetc(
   int c,
   FILE *stream
);
wint_t ungetwc(
   wint_t c,
   FILE *stream
);
```

Parameters

c
Character to be pushed.

stream
Pointer to **FILE** structure.

Return Value

If successful, each of these functions returns the character argument *c*. If *c* cannot be pushed back or if no character has been read, the input stream is unchanged and **ungetc** returns **EOF**; **ungetwc** returns **WEOF**. If *stream* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **EOF** or **WEOF** is returned and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

The **ungetc** function pushes the character *c* back onto *stream* and clears the end-of-file indicator. The stream must be open for reading. A subsequent read operation on *stream* starts with *c*. An attempt to push **EOF** onto the stream using **ungetc** is ignored.

Characters placed on the stream by **ungetc** may be erased if **fflush**, fseek, **fsetpos**, or rewind is called before the character is read from the stream. The file-position indicator will have the value it had before the characters were pushed back. The external storage corresponding to the stream is unchanged. On a successful **ungetc** call against a text stream, the file-position indicator is unspecified until all the pushed-back characters are read or discarded. On each successful **ungetc** call against a binary stream, the file-position indicator is decremented; if its value was 0 before a call, the value is undefined after the call.

Results are unpredictable if **ungetc** is called twice without a read or file-positioning operation between the two calls. After a call to **fscanf**, a call to **ungetc** may fail unless another read operation (such as **getc**) has been performed. This is because **fscanf** itself calls **ungetc**.

ungetwc is a wide-character version of **ungetc**. However, on each successful **ungetwc** call against a text or binary stream, the value of the file-position indicator is unspecified until all pushed-back characters are read or discarded.

These functions are thread-safe and lock sensitive data during execution. For a non-locking version, see _ungetc_nolock, _ungetwc_nolock.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ungettc	**ungetc**	**ungetc**	**ungetwc**

Requirements

ROUTINE	REQUIRED HEADER
ungetc	<stdio.h>
ungetwc	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_ungetc.c
// This program first converts a character
// representation of an unsigned integer to an integer. If
// the program encounters a character that is not a digit,
// the program uses ungetc to replace it in the  stream.
//

#include <stdio.h>
#include <ctype.h>

int main( void )
{
   int ch;
   int result = 0;

   // Read in and convert number:
   while( ((ch = getchar()) != EOF) && isdigit( ch ) )
      result = result * 10 + ch - '0';    // Use digit.
   if( ch != EOF )
      ungetc( ch, stdin );                // Put nondigit back.
   printf( "Number = %d\nNext character in stream = '%c'",
            result, getchar() );
}
```

```
      521aNumber = 521
Next character in stream = 'a'
```

See also

Stream I/O
getc, getwc
putc, putwc

_ungetc_nolock, _ungetwc_nolock

11/8/2018 • 2 minutes to read • Edit Online

Pushes a character back onto the stream.

Syntax

```
int _ungetc_nolock(
   int c,
   FILE *stream
);
wint_t _ungetwc_nolock(
   wint_t c,
   FILE *stream
);
```

Parameters

c
Character to be pushed.

stream
Pointer to **FILE** structure.

Return Value

If successful, each of these functions returns the character argument *c*. If *c* cannot be pushed back or if no character has been read, the input stream is unchanged and **_ungetc_nolock** returns **EOF**; **_ungetwc_nolock** returns **WEOF**. If *stream* is **NULL**, **EOF** or **WEOF** is returned and **errno** is set to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

These functions are non-locking versions of **ungetc** and **ungetwc**. The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ungettc_nolock	**_ungetc_nolock**	**_ungetc_nolock**	**_ungetwc_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_ungetc_nolock	<stdio.h>

ROUTINE	REQUIRED HEADER
_ungetwc_nolock	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

See also

Stream I/O
getc, getwc
putc, putwc

ungetch

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _ungetch instead.

IMPORTANT

This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

_ungetch, _ungetwch, _ungetch_nolock, _ungetwch_nolock

10/31/2018 • 2 minutes to read • Edit Online

Pushes back the last character that's read from the console.

> **IMPORTANT**
>
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _ungetch(
   int c
);
wint_t _ungetwch(
   wint_t c
);
int _ungetch_nolock(
   int c
);
wint_t _ungetwch_nolock(
   wint_t c
);
```

Parameters

c
Character to be pushed.

Return Value

Both functions return the character *c* if successful. If there is an error, **_ungetch** returns a value of **EOF** and **_ungetwch** returns **WEOF**.

Remarks

These functions push the character *c* back to the console, causing *c* to be the next character read by **_getch** or **_getche** (or **_getwch** or **_getwche**). **_ungetch** and **_ungetwch** fail if they are called more than once before the next read. The *c* argument may not be **EOF** (or **WEOF**).

The versions with the **_nolock** suffix are identical except that they are not protected from interference by other threads. They may be faster since they do not incur the overhead of locking out other threads. Use these functions only in thread-safe contexts such as single-threaded applications or where the calling scope already handles thread isolation.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_ungettch	**_ungetch**	**_ungetch**	**_ungetwch**
_ungettch_nolock	**_ungetch_nolock**	**_ungetch_nolock**	**_ungetwch_nolock**

Requirements

ROUTINE	REQUIRED HEADER
_ungetch, **_ungetch_nolock**	<conio.h>
_ungetwch, **_ungetwch_nolock**	<conio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_ungetch.c
// compile with: /c
// In this program, a white-space delimited
// token is read from the keyboard. When the program
// encounters a delimiter, it uses _ungetch to replace
// the character in the keyboard buffer.
//

#include <conio.h>
#include <ctype.h>
#include <stdio.h>

int main( void )
{
   char buffer[100];
   int count = 0;
   int ch;

   ch = _getche();
   while( isspace( ch ) )      // Skip preceding white space.
      ch = _getche();
   while( count < 99 )         // Gather token.
   {
      if( isspace( ch ) )      // End of token.
         break;
      buffer[count++] = (char)ch;
      ch = _getche();
   }
   _ungetch( ch );            // Put back delimiter.
   buffer[count] = '\0';      // Null terminate the token.
   printf( "\ntoken = %s\n", buffer );
}
```

```
Whitetoken = White
```

See also

Console and Port I/O

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l

_getch, _getwch

unlink

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _unlink instead.

_unlink, _wunlink

1/21/2019 • 2 minutes to read • Edit Online

Delete a file.

Syntax

```
int _unlink(
   const char *filename
);
int _wunlink(
   const wchar_t *filename
);
```

Parameters

filename
Name of file to remove.

Return Value

Each of these functions returns 0 if successful. Otherwise, the function returns -1 and sets **errno** to **EACCES**, which means the path specifies a read-only file or a directory, or to **ENOENT**, which means the file or path is not found.

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

Remarks

The **_unlink** function deletes the file specified by *filename*. **_wunlink** is a wide-character version of **_unlink**; the *filename* argument to **_wunlink** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tunlink	**_unlink**	**_unlink**	**_wunlink**

Requirements

ROUTINE	REQUIRED HEADER
_unlink	<io.h> and <stdio.h>
_wunlink	<io.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Code Example

This program uses _unlink to delete CRT_UNLINK.TXT.

```
// crt_unlink.c

#include <stdio.h>

int main( void )
{
   if( _unlink( "crt_unlink.txt" ) == -1 )
      perror( "Could not delete 'CRT_UNLINK.TXT'" );
   else
      printf( "Deleted 'CRT_UNLINK.TXT'\n" );
}
```

Input: crt_unlink.txt

```
This file will be deleted.
```

Sample Output

```
Deleted 'CRT_UNLINK.TXT'
```

See also

File Handling
_close
remove, _wremove

_unlock_file

10/31/2018 • 2 minutes to read • Edit Online

Unlocks a file, allowing other processes to access the file.

Syntax

```
void _unlock_file(
   FILE* file
);
```

Parameters

file
File handle.

Remarks

The **_unlock_file** function unlocks the file specified by *file*. Unlocking a file allows access to the file by other processes. This function should not be called unless **_lock_file** was previously called on the *file* pointer. Calling **_unlock_file** on a file that isn't locked may result in a deadlock. For an example, see _lock_file.

Requirements

ROUTINE	REQUIRED HEADER
_unlock_file	<stdio.h>

For additional compatibility information, see Compatibility.

See also

File Handling
_creat, _wcreat
_open, _wopen
_lock_file

_utime, _utime32, _utime64, _wutime, _wutime32, _wutime64

11/9/2018 • 3 minutes to read • Edit Online

Set the file modification time.

Syntax

```
int _utime(
   const char *filename,
   struct _utimbuf *times
);
int _utime32(
   const char *filename,
   struct __utimbuf32 *times
);
int _utime64(
   const char *filename,
   struct __utimbuf64 *times
);
int _wutime(
   const wchar_t *filename,
   struct _utimbuf *times
);
int _wutime32(
   const wchar_t *filename,
   struct __utimbuf32 *times
);
int _wutime64(
   const wchar_t *filename,
   struct __utimbuf64 *times
);
```

Parameters

filename
Pointer to a string that contains the path or filename.

times
Pointer to stored time values.

Return Value

Each of these functions returns 0 if the file-modification time was changed. A return value of -1 indicates an error. If an invalid parameter is passed, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and **errno** is set to one of the following values:

ERRNO VALUE	CONDITION
EACCES	Path specifies directory or read-only file
EINVAL	Invalid *times* argument

ERRNO VALUE	CONDITION
EMFILE	Too many open files (the file must be opened to change its modification time)
ENOENT	Path or filename not found

See _doserrno, errno, _sys_errlist, and _sys_nerr for more information on these, and other, return codes.

The date can be changed for a file if the change date is after midnight, January 1, 1970, and before the end date of the function used. **_utime** and **_wutime** use a 64-bit time value, so the end date is 23:59:59, December 31, 3000, UTC. If **_USE_32BIT_TIME_T** is defined to force the old behavior, the end date is 23:59:59 January 18, 2038, UTC. **_utime32** or **_wutime32** use a 32-bit time type regardless of whether **_USE_32BIT_TIME_T** is defined, and always have the earlier end date. **_utime64** or **_wutime64** always use the 64-bit time type, so these functions always support the later end date.

Remarks

The **_utime** function sets the modification time for the file specified by *filename*. The process must have write access to the file in order to change the time. In the Windows operating system, you can change the access time and the modification time in the **_utimbuf** structure. If *times* is a **NULL** pointer, the modification time is set to the current local time. Otherwise, *times* must point to a structure of type **_utimbuf**, defined in SYS\UTIME.H.

The **_utimbuf** structure stores file access and modification times used by **_utime** to change file-modification dates. The structure has the following fields, which are both of type **time_t**:

FIELD	
actime	Time of file access
modtime	Time of file modification

Specific versions of the **_utimbuf** structure (**_utimebuf32** and **__utimbuf64**) are defined using the 32-bit and 64-bit versions of the time type. These are used in the 32-bit and 64-bit specific versions of this function. **_utimbuf** itself by default uses a 64-bit time type unless **_USE_32BIT_TIME_T** is defined.

_utime is identical to **_futime** except that the *filename* argument of **_utime** is a filename or a path to a file, rather than a file descriptor of an open file.

_wutime is a wide-character version of **_utime**; the *filename* argument to **_wutime** is a wide-character string. These functions behave identically otherwise.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_tutime	**_utime**	**_utime**	**_wutime**
_tutime32	**_utime32**	**_utime32**	**_wutime32**
_tutime64	**_utime64**	**_utime64**	**_wutime64**

Requirements

ROUTINE	REQUIRED HEADERS	OPTIONAL HEADERS
_utime, **_utime32**, **_utime64**	<sys/utime.h>	<errno.h>
_utime64	<sys/utime.h>	<errno.h>
_wutime	<utime.h> or <wchar.h>	<errno.h>

For additional compatibility information, see Compatibility.

Example

This program uses **_utime** to set the file-modification time to the current time.

```
// crt_utime.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utime.h>
#include <time.h>

int main( void )
{
   struct tm tma = {0}, tmm = {0};
   struct _utimbuf ut;

   // Fill out the accessed time structure
   tma.tm_hour = 12;
   tma.tm_isdst = 0;
   tma.tm_mday = 15;
   tma.tm_min = 0;
   tma.tm_mon = 0;
   tma.tm_sec = 0;
   tma.tm_year = 103;

   // Fill out the modified time structure
   tmm.tm_hour = 12;
   tmm.tm_isdst = 0;
   tmm.tm_mday = 15;
   tmm.tm_min = 0;
   tmm.tm_mon = 0;
   tmm.tm_sec = 0;
   tmm.tm_year = 102;

   // Convert tm to time_t
   ut.actime = mktime(&tma);
   ut.modtime = mktime(&tmm);

   // Show file time before and after
   system( "dir crt_utime.c" );
   if( _utime( "crt_utime.c", &ut ) == -1 )
      perror( "_utime failed\n" );
   else
      printf( "File time modified\n" );
   system( "dir crt_utime.c" );
}
```

Sample Output

```
Volume in drive C has no label.
Volume Serial Number is 9CAC-DE74

Directory of C:\test

01/09/2003  05:38 PM               935 crt_utime.c
               1 File(s)            935 bytes
               0 Dir(s)  20,742,955,008 bytes free
File time modified
Volume in drive C has no label.
Volume Serial Number is 9CAC-DE74

Directory of C:\test

01/15/2002  12:00 PM               935 crt_utime.c
               1 File(s)            935 bytes
               0 Dir(s)  20,742,955,008 bytes free
```

See also

Time Management
asctime, _wasctime
ctime, _ctime32, _ctime64, _wctime, _wctime32, _wctime64
_fstat, _fstat32, _fstat64, _fstati64, _fstat32i64, _fstat64i32
_ftime, _ftime32, _ftime64
_futime, _futime32, _futime64
gmtime, _gmtime32, _gmtime64
localtime, _localtime32, _localtime64
_stat, _wstat Functions
time, _time32, _time64

va_arg, va_copy, va_end, va_start

10/31/2018 • 4 minutes to read • Edit Online

Accesses variable-argument lists.

Syntax

```
type va_arg(
   va_list arg_ptr,
   type
);
void va_copy(
   va_list dest,
   va_list src
); // (ISO C99 and later)
void va_end(
   va_list arg_ptr
);
void va_start(
   va_list arg_ptr,
   prev_param
); // (ANSI C89 and later)
void va_start(
   arg_ptr
);  // (deprecated Pre-ANSI C89 standardization version)
```

Parameters

type
Type of argument to be retrieved.

arg_ptr
Pointer to the list of arguments.

dest
Pointer to the list of arguments to be initialized from *src*

src
Pointer to the initialized list of arguments to copy to *dest*.

prev_param
Parameter that precedes the first optional argument.

Return Value

va_arg returns the current argument. **va_copy**, **va_start** and **va_end** do not return values.

Remarks

The **va_arg**, **va_copy**, **va_end**, and **va_start** macros provide a portable way to access the arguments to a function when the function takes a variable number of arguments. There are two versions of the macros: The macros defined in STDARG.H conform to the ISO C99 standard; the macros defined in VARARGS.H are deprecated but are retained for backward compatibility with code that was written before the ANSI C89 standard.

These macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be accessed through the parameter names. The optional arguments are accessed through the macros in STDARG.H (or VARARGS.H for code that was written before the ANSI C89 standard), which sets a pointer to the first optional argument in the argument list, retrieves arguments from the list, and resets the pointer when argument processing is completed.

The C standard macros, defined in STDARG.H, are used as follows:

- **va_start** sets *arg_ptr* to the first optional argument in the list of arguments that's passed to the function. The argument *arg_ptr* must have the **va_list** type. The argument *prev_param* is the name of the required parameter that immediately precedes the first optional argument in the argument list. If *prev_param* is declared with the register storage class, the macro's behavior is undefined. **va_start** must be used before **va_arg** is used for the first time.
- **va_arg** retrieves a value of *type* from the location that's given by *arg_ptr*, and increments *arg_ptr* to point to the next argument in the list by using the size of *type* to determine where the next argument starts. **va_arg** can be used any number of times in the function to retrieve arguments from the list.
- **va_copy** makes a copy of a list of arguments in its current state. The *src* parameter must already be initialized with **va_start**; it may have been updated with **va_arg** calls, but must not have been reset with **va_end**. The next argument that's retrieved by **va_arg** from *dest* is the same as the next argument that's retrieved from *src*.
- After all arguments have been retrieved, **va_end** resets the pointer to **NULL**. **va_end** must be called on each argument list that's initialized with **va_start** or **va_copy** before the function returns.

> **NOTE**
> The macros in VARARGS.H are deprecated and are retained only for backwards compatibility with code that was written before the ANSI C89 standard. In all other cases, use the macros in STDARGS.H.

When they are compiled by using /clr (Common Language Runtime Compilation), programs that use these macros may generate unexpected results because of differences between native and common language runtime (CLR) type systems. Consider this program:

```
#include <stdio.h>
#include <stdarg.h>

void testit (int i, ...)
{
    va_list argptr;
    va_start(argptr, i);

    if (i == 0)
    {
        int n = va_arg(argptr, int);
        printf("%d\n", n);
    }
    else
    {
        char *s = va_arg(argptr, char*);
        printf("%s\n", s);
    }

    va_end(argptr);
}

int main()
{
    testit(0, 0xFFFFFFFF); // 1st problem: 0xffffffff is not an int
    testit(1, NULL);       // 2nd problem: NULL is not a char*
}
```

Notice that **testit** expects its second parameter to be either an **int** or a **char***. The arguments being passed are 0xffffffff (an **unsigned int**, not an **int**) and **NULL** (actually an **int**, not a **char***). When the program is compiled for native code, it produces this output:

```
-1

(null)
```

Requirements

Header: <stdio.h> and <stdarg.h>

Deprecated Header: <varargs.h>

Libraries

All versions of the C run-time libraries.

Example

```
// crt_va.c
// Compile with: cl /W3 /Tc crt_va.c
// The program below illustrates passing a variable
// number of arguments using the following macros:
//      va_start            va_arg              va_copy
//      va_end              va_list

#include <stdio.h>
#include <stdarg.h>
#include <math.h>

double deviation(int first, ...);

int main( void )
{
    /* Call with 3 integers (-1 is used as terminator). */
    printf("Deviation is: %f\n", deviation(2, 3, 4, -1 ));

    /* Call with 4 integers. */
    printf("Deviation is: %f\n", deviation(5, 7, 9, 11, -1));

    /* Call with just -1 terminator. */
    printf("Deviation is: %f\n", deviation(-1));
}

/* Returns the standard deviation of a variable list of integers. */
double deviation(int first, ...)
{
    int count = 0, i = first;
    double mean = 0.0, sum = 0.0;
    va_list marker;
    va_list copy;

    va_start(marker, first);     /* Initialize variable arguments. */
    va_copy(copy, marker);       /* Copy list for the second pass */
    while (i != -1)
    {
        sum += i;
        count++;
        i = va_arg(marker, int);
    }
    va_end(marker);              /* Reset variable argument list. */
    mean = sum ? (sum / count) : 0.0;

    i = first;                  /* reset to calculate deviation */
    sum = 0.0;
    while (i != -1)
    {
        sum += (i - mean)*(i - mean);
        i = va_arg(copy, int);
    }
    va_end(copy);               /* Reset copy of argument list. */
    return count ? sqrt(sum / count) : 0.0;
}
```

```
Deviation is: 0.816497
Deviation is: 2.236068
Deviation is: 0.000000
```

See also

Argument Access
vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

_vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output to the console by using a pointer to a list of arguments. More secure versions of these functions are available, see _vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _vcprintf(
   const char* format,
   va_list argptr
);
int _vcprintf_l(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

The number of characters written, or a negative value if an output error occurs. If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to the console.

_vcwprintf is the wide-character version of **_vcprintf**. It takes a wide-character string as an argument.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtcprintf	**_vcprintf**	**_vcprintf**	**_vcwprintf**
_vtcprintf_l	**_vcprintf_l**	**_vcprintf_l**	**_vcwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_vcprintf, **_vcprintf_l**	<conio.h> and <stdarg.h>	<varargs.h>*
_vcwprintf, **_vcwprintf_l**	<conio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt_vcprintf.cpp
// compile with: /c
#include <conio.h>
#include <stdarg.h>

// An error formatting function used to print to the console.
int eprintf(const char* format, ...)
{
    va_list args;
    va_start(args, format);
    int result = _vcprintf(format, args);
    va_end(args);
    return result;
}

int main()
{
    eprintf("(%d:%d): Error %s%d : %s\n", 10, 23, "C", 2111,
           "<some error text>");
    eprintf("    (Related to symbol '%s' defined on line %d).\n",
            "<symbol>", 5 );
}
```

```
(10,23): Error C2111 : <some error text>
    (Related to symbol '<symbol>' defined on line 5).
```

See also

Stream I/O
vprintf Functions
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

_vcprintf_p, _vcprintf_p_l, _vcwprintf_p, _vcwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output to the console by using a pointer to a list of arguments, and supports positional parameters in the format string.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _vcprintf_p(
   const char* format,
   va_list argptr
);
int _vcprintf_p_l(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf_p(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_p_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
The format specification.

argptr
A pointer to a list of arguments.

locale
The locale to use.

For more information, see Format Specification Syntax: printf and wprintf Functions.

Return Value

The number of characters that are written, or a negative value if an output error occurs. If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and -1 is returned.

Remarks

Each of these functions takes a pointer to an argument list, and then uses the **_putch** function to format and write

the given data to the console. (**_vcwprintf_p** uses **_putwch** instead of **_putch**. **_vcwprintf_p** is the wide-character version of **_vcprintf_p**. It takes a wide-character string as an argument.)

The versions of these functions that have the **_l** suffix are identical except that they use the locale parameter that's passed in instead of the current locale.

Each *argument* (if any) is converted and is output according to the corresponding format specification in *format*. The format specification supports positional parameters so that you can specify the order in which the arguments are used in the format string. For more information, see printf_p Positional Parameters.

These functions do not translate line-feed characters into carriage return-line feed (CR-LF) combinations when they are output.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

These functions validate the input pointer and the format string. If *format* or *argument* is **NULL**, or if the format string contains invalid formatting characters, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE AND _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtcprintf_p	**_vcprintf_p**	**_vcprintf_p**	**_vcwprintf_p**
_vtcprintf_p_l	**_vcprintf_p_l**	**_vcprintf_p_l**	**_vcwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_vcprintf_p, **_vcprintf_p_l**	<conio.h> and <stdarg.h>
_vcwprintf_p, **_vcwprintf_p_l**	<conio.h> and <stdarg.h>

For more compatibility information, see Compatibility.

Example

```
// crt_vcprintf_p.c
// compile with: /c
#include <conio.h>
#include <stdarg.h>

// An error formatting function that's used to print to the console.
int eprintf(const char* format, ...)
{
   va_list args;
   va_start(args, format);
   int result = _vcprintf_p(format, args);
   va_end(args);
   return result;
}

int main()
{
   int n = eprintf("parameter 2 = %2$d; parameter 1 = %1$s\r\n",
      "one", 222);
   _cprintf_s("%d characters printed\r\n");
}
```

```
parameter 2 = 222; parameter 1 = one
38 characters printed
```

See also

Console and Port I/O
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
va_arg, va_copy, va_end, va_start
printf_p Positional Parameters

_vcprintf_s, _vcprintf_s_l, _vcwprintf_s, _vcwprintf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output to the console by using a pointer to a list of arguments. These versions of _vcprintf, _vcprintf_l, _vcwprintf, _vcwprintf_l have security enhancements, as described in Security Features in the CRT.

> **IMPORTANT**
> This API cannot be used in applications that execute in the Windows Runtime. For more information, see CRT functions not supported in Universal Windows Platform apps.

Syntax

```
int _vcprintf(
   const char* format,
   va_list argptr
);
int _vcprintf(
   const char* format,
   locale_t locale,
   va_list argptr
);
int _vcwprintf_s(
   const wchar_t* format,
   va_list argptr
);
int _vcwprintf_s_l(
   const wchar_t* format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format specification.

argptr
Pointer to the list of arguments.

locale
The locale to use.

For more information, see Format Specification Syntax: printf and wprintf Functions.

Return Value

The number of characters written, or a negative value if an output error occurs.

Like the less secure versions of these functions, if *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. Additionally, unlike the less secure versions of these functions, if *format* does not specify a valid format, an invalid parameter exception is generated. If execution is allowed to continue, these functions return an error code and set **errno** to that error code. The default error code is **EINVAL** if a more specific value does not apply.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the console. **_vcwprintf_s** is the wide-character version of **_vcprintf_s**. It takes a wide-character string as an argument.

The versions of these functions that have the **_l** suffix are identical except that they use the locale parameter that's passed in instead of the current locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtcprintf_s	**_vcprintf_s**	**_vcprintf_s**	**_vcwprintf_s**
_vtcprintf_s_l	**_vcprintf_s_l**	**_vcprintf_s_l**	**_vcwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_vcprintf_s, **_vcprintf_s_l**	<conio.h> and <stdarg.h>	<varargs.h>*
_vcwprintf_s, **_vcwprintf_s_l**	<conio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt_vcprintf_s.cpp
#include <conio.h>
#include <stdarg.h>

// An error formatting function used to print to the console.
int eprintf_s(const char* format, ...)
{
    va_list args;
    va_start(args, format);
    int result = _vcprintf_s(format, args);
    va_end(args);
    return result;
}

int main()
{
    eprintf_s("(%d:%d): Error %s%d : %s\n", 10, 23, "C", 2111,
              "<some error text>");
    eprintf_s("    (Related to symbol '%s' defined on line %d).\n",
              "<symbol>", 5 );
}
```

```
(10,23): Error C2111 : <some error text>
    (Related to symbol '<symbol>' defined on line 5).
```

See also

Stream I/O
vprintf Functions
_cprintf, _cprintf_l, _cwprintf, _cwprintf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. More secure versions of these functions exist; see vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l.

Syntax

```
int vfprintf(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_l(
   FILE *stream,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vfwprintf(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
int _vfwprintf_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

vfprintf and **vfwprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If either *stream* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*.

vfwprintf is the wide-character version of **vfprintf**; the two functions behave identically if the stream is opened in ANSI mode. **vfprintf** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vftprintf	**vfprintf**	**vfprintf**	**vfwprintf**
_vftprintf_l	**_vfprintf_l**	**_vfprintf_l**	**_vfwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vfprintf, **_vfprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vfwprintf, **_vfwprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments, with the ability to specify the order that arguments are used in the format string.

Syntax

```
int _vfprintf_p(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_p_l(
   FILE *stream,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vfwprintf_p(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
int _vfwprintf_p_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

_vfprintf_p and **_vfwprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*. These functions differ from the **_vfprint_s** and **_vfwprint_s** versions only in that they support positional

parameters. For more information, see printf_p Positional Parameters.

_vfwprintf_p is the wide-character version of **_vprintf_p**; the two functions behave identically if the stream is opened in ANSI mode. **_vprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

If either *stream* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vftprintf_p	**_vfprintf_p**	**_vfprintf_p**	**_vfwprintf_p**
_vftprintf_p_l	**_vfprintf_p_l**	**_vfprintf_p_l**	**_vfwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_vfprintf_p, **_vfprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
_vfwprintf_p, **_vfwprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start
printf_p Positional Parameters
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l

vfprintf_s, _vfprintf_s_l, vfwprintf_s, _vfwprintf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. These are versions of vfprintf, _vfprintf_l, vfwprintf, _vfwprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int vfprintf_s(
   FILE *stream,
   const char *format,
   va_list argptr
);
int _vfprintf_s_l(
   FILE *stream,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vfwprintf_s(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
int _vfwprintf_s_l(
   FILE *stream,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

vfprintf_s and **vfwprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If either *stream* or *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to *stream*.

These functions differ from the non-secure versions only in that the secure versions check that the *format* string contains valid formatting characters.

vfwprintf_s is the wide-character version of **vfprintf_s**; the two functions behave identically if the stream is opened in ANSI mode. **vfprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
>
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vftprintf_s	**vfprintf_s**	**vfprintf_s**	**vfwprintf_s**
_vftprintf_s_l	**_vfprintf_s_l**	**_vfprintf_s_l**	**_vfwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vfprintf_s, **_vfprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vfwprintf_s, **_vfwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vfscanf, vfwscanf

11/9/2018 • 2 minutes to read • Edit Online

Reads formatted data from a stream. More secure versions of these functions are available; see vfscanf_s, vfwscanf_s.

Syntax

```
int vfscanf(
   FILE *stream,
   const char *format,
   va_list argptr
);
int vfwscanf(
   FILE *stream,
   const wchar_t *format,
   va_list argptr
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

arglist
Variable argument list.

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that are read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **vfscanf** and **vfwscanf**.

These functions validate their parameters. If *stream* or *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **vfscanf** function reads data from the current position of *stream* into the locations that are given by the *arglist* argument list. Each argument in the list must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf**; see scanf for a description of *format*.

vfwscanf is a wide-character version of **vfscanf**; the format argument to **vfwscanf** is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **vfscanf** doesn't support input from a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vftscanf	**vfscanf**	**vfscanf**	**vfwscanf**

For more information, see Format Specification Fields: scanf and wscanf Functions.

Requirements

FUNCTION	REQUIRED HEADER
vfscanf	<stdio.h>
vfwscanf	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_vfscanf.c
// compile with: /W3
// This program writes formatted
// data to a file. It then uses vfscanf to
// read the various data back from the file.

#include <stdio.h>
#include <stdarg.h>

FILE *stream;

int call_vfscanf(FILE * istream, char * format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vfscanf(istream, format, arglist);
    va_end(arglist);
    return result;
}

int main(void)
{
    long l;
    float fp;
    char s[81];
    char c;

    if (fopen_s(&stream, "vfscanf.out", "w+") != 0)
    {
        printf("The file vfscanf.out was not opened\n");
    }
    else
    {
        fprintf(stream, "%s %ld %f%c", "a-string",
            65000, 3.14159, 'x');
        // Security caution!
        // Beware loading data from a file without confirming its size,
        // as it may lead to a buffer overrun situation.

        // Set pointer to beginning of file:
        fseek(stream, 0L, SEEK_SET);

        // Read data back from file:
        call_vfscanf(stream, "%s %ld %f%c", s, &l, &fp, &c);

        // Output data read:
        printf("%s\n", s);
        printf("%ld\n", l);
        printf("%f\n", fp);
        printf("%c\n", c);

        fclose(stream);
    }
}
```

```
a-string
65000
3.141590
x
```

See also

Stream I/O

_cscanf, _cscanf_l, _cwscanf, _cwscanf_l
fprintf, _fprintf_l, fwprintf, _fwprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
fscanf_s, _fscanf_s_l, fwscanf_s, _fwscanf_s_l
vfscanf_s, vfwscanf_s

vfscanf_s, vfwscanf_s

11/9/2018 • 2 minutes to read • Edit Online

Reads formatted data from a stream. These versions of vfscanf, vfwscanf have security enhancements, as described in Security Features in the CRT.

Syntax

```
int vfscanf_s(
   FILE *stream,
   const char *format,
   va_list arglist
);
int vfwscanf_s(
   FILE *stream,
   const wchar_t *format,
   va_list arglist
);
```

Parameters

stream
Pointer to **FILE** structure.

format
Format-control string.

arglist
Variable argument list.

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If an error occurs, or if the end of the file stream is reached before the first conversion, the return value is **EOF** for **vfscanf_s** and **vfwscanf_s**.

These functions validate their parameters. If *stream* is an invalid file pointer, or *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

Remarks

The **vfscanf_s** function reads data from the current position of *stream* into the locations that are given by the *arglist* argument list (if any). Each argument in the list must be a pointer to a variable of a type that corresponds to a type specifier in *format*. *format* controls the interpretation of the input fields and has the same form and function as the *format* argument for **scanf_s**; see Format Specification Fields: scanf and wscanf Functions for a description of *format*. **vfwscanf_s** is a wide-character version of **vfscanf_s**; the format argument to **vfwscanf_s** is a wide-character string. These functions behave identically if the stream is opened in ANSI mode. **vfscanf_s** doesn't currently support input from a UNICODE stream.

The main difference between the more secure functions (that have the **_s** suffix) and the other versions is that the more secure functions require the size in characters of each **c**, **C**, **s**, **S**, and **[** type field to be passed as an argument

immediately following the variable. For more information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Width Specification.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vftscanf_s	**vfscanf_s**	**vfscanf_s**	**vfwscanf_s**

Requirements

FUNCTION	REQUIRED HEADER
vfscanf_s	<stdio.h>
vfwscanf_s	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_vfscanf_s.c
// compile with: /W3
// This program writes formatted
// data to a file. It then uses vfscanf_s to
// read the various data back from the file.

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

FILE *stream;

int call_vfscanf_s(FILE * istream, char * format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vfscanf_s(istream, format, arglist);
    va_end(arglist);
    return result;
}

int main(void)
{
    long l;
    float fp;
    char s[81];
    char c;

    if (fopen_s(&stream, "vfscanf_s.out", "w+") != 0)
    {
        printf("The file vfscanf_s.out was not opened\n");
    }
    else
    {
        fprintf(stream, "%s %ld %f%c", "a-string",
            65000, 3.14159, 'x');
        // Security caution!
        // Beware loading data from a file without confirming its size,
        // as it may lead to a buffer overrun situation.

        // Set pointer to beginning of file:
        fseek(stream, 0L, SEEK_SET);

        // Read data back from file:
        call_vfscanf_s(stream, "%s %ld %f%c", s, _countof(s), &l, &fp, &c, 1);

        // Output data read:
        printf("%s\n", s);
        printf("%ld\n", l);
        printf("%f\n", fp);
        printf("%c\n", c);

        fclose(stream);
    }
}
```

```
a-string
65000
3.141590
x
```

See also

Stream I/O

_cscanf_s, _cscanf_s_l, _cwscanf_s, _cwscanf_s_l

fprintf_s, _fprintf_s_l, fwprintf_s, _fwprintf_s_l

scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l

sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l

fscanf, _fscanf_l, fwscanf, _fwscanf_l

vfscanf, vfwscanf

vprintf, _vprintf_l, vwprintf, _vwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output by using a pointer to a list of arguments. More secure versions of these functions are available, see vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l.

Syntax

```
int vprintf(
   const char *format,
   va_list argptr
);
int _vprintf_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int vwprintf(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

vprintf and **vwprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**.

vwprintf is the wide-character version of **vprintf**; the two functions behave identically if the stream is opened in ANSI mode. **vprintf** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
>
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns. Note that invalid format strings are detected and result in an error.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtprintf	**vprintf**	**vprintf**	**vwprintf**
_vtprintf_l	**_vprintf_l**	**_vprintf_l**	**_vwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vprintf, **_vprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vwprintf, **_vwprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

_vprintf_p, _vprintf_p_l, _vwprintf_p, _vwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output by using a pointer to a list of arguments, and enables specification of the order in which the arguments are used.

Syntax

```
int _vprintf_p(
   const char *format,
   va_list argptr
);
int _vprintf_p_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vwprintf_p(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_p_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

_vprintf_p and **_vwprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**. These functions differ from **vprintf_s** and **vwprintf_s** only in that they support the ability to specify the order in which the arguments are used. For more information, see printf_p Positional Parameters.

_vwprintf_p is the wide-character version of **_vprintf_p**; the two functions behave identically if the stream is opened in ANSI mode. **_vprintf_p** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in

instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtprintf_p	**_vprintf_p**	**_vprintf_p**	**_vwprintf_p**
_vtprintf_p_l	**_vprintf_p_l**	**_vprintf_p_l**	**_vwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_vprintf_p, **_vprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
_vwprintf_p, **_vwprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
_fprintf_p, _fprintf_p_l, _fwprintf_p, _fwprintf_p_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
_sprintf_p, _sprintf_p_l, _swprintf_p, _swprintf_p_l
vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l
va_arg, va_copy, va_end, va_start
_vfprintf_p, _vfprintf_p_l, _vfwprintf_p, _vfwprintf_p_l
_printf_p, _printf_p_l, _wprintf_p, _wprintf_p_l
printf_p Positional Parameters

vprintf_s, _vprintf_s_l, vwprintf_s, _vwprintf_s_l

10/31/2018 • 2 minutes to read • Edit Online

Writes formatted output by using a pointer to a list of arguments. These versions of vprintf, _vprintf_l, vwprintf, _vwprintf_l have security enhancements, as described in Security Features in the CRT.

Syntax

```
int vprintf_s(
   const char *format,
   va_list argptr
);
int _vprintf_s_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int vwprintf_s(
   const wchar_t *format,
   va_list argptr
);
int _vwprintf_s_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

vprintf_s and **vwprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *format* is a null pointer, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes the given data to **stdout**.

The secure versions of these functions differ from **vprintf** and **vwprintf** only in that the secure versions check that the format string contains valid formatting characters.

vwprintf_s is the wide-character version of **vprintf_s**; the two functions behave identically if the stream is opened in ANSI mode. **vprintf_s** doesn't currently support output into a UNICODE stream.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtprintf_s	**vprintf_s**	**vprintf_s**	**vwprintf_s**
_vtprintf_s_l	**_vprintf_s_l**	**_vprintf_s_l**	**_vwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vprintf_s, **_vprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vwprintf_s, **_vwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vscanf, vwscanf

11/9/2018 • 2 minutes to read • Edit Online

Reads formatted data from the standard input stream. More secure versions of these function are available; see vscanf_s, vwscanf_s.

Syntax

```
int vscanf(
   const char *format,
   va_list arglist
);
int vwscanf(
   const wchar_t *format,
   va_list arglist
);
```

Parameters

format
Format control string.

arglist
Variable argument list.

Return Value

Returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned.

If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **vscanf** function reads data from the standard input stream **stdin** and writes the data into the locations that are given by the *arglist* argument list. Each argument in the list must be a pointer to a variable of a type that corresponds to a type specifier in *format*. If copying occurs between strings that overlap, the behavior is undefined.

> **IMPORTANT**
> When you use **vscanf** to read a string, always specify a width for the **%s** format (for example, **"%32s"** instead of **"%s"**); otherwise, incorrectly formatted input can cause a buffer overrun. As an alternative, you can use vscanf_s, vwscanf_s or fgets.

vwscanf is a wide-character version of **vscanf**; the *format* argument to **vwscanf** is a wide-character string. **vwscanf** and **vscanf** behave identically if the stream is opened in ANSI mode. **vscanf** doesn't support input from a UNICODE stream.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtscanf	**vscanf**	**vscanf**	**vwscanf**

For more information, see Format Specification Fields: scanf and wscanf Functions.

Requirements

ROUTINE	REQUIRED HEADER
vscanf	<stdio.h>
vwscanf	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_vscanf.c
// compile with: /W3
// This program uses the vscanf and vwscanf functions
// to read formatted input.

#include <stdio.h>
#include <stdarg.h>

int call_vscanf(char *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vscanf(format, arglist);
    va_end(arglist);
    return result;
}

int call_vwscanf(wchar_t *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vwscanf(format, arglist);
    va_end(arglist);
    return result;
}

int main( void )
{
    int   i, result;
    float fp;
    char  c, s[81];
    wchar_t wc, ws[81];
    result = call_vscanf( "%d %f %c %C %80s %80S", &i, &fp, &c, &wc, s, ws );
    printf( "The number of fields input is %d\n", result );
    printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c, wc, s, ws);
    result = call_vwscanf( L"%d %f %hc %lc %80S %80ls", &i, &fp, &c, &wc, s, ws );
    wprintf( L"The number of fields input is %d\n", result );
    wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp, c, wc, s, ws);
}
```

```
      71 98.6 h z Byte characters
36 92.3 y n Wide charactersThe number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

See also

Floating-Point Support
Stream I/O
Locale
fscanf, _fscanf_l, fwscanf, _fwscanf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vscanf_s, vwscanf_s

vscanf_s, vwscanf_s

10/31/2018 • 3 minutes to read • Edit Online

Reads formatted data from the standard input stream. These versions of vscanf, vwscanf have security enhancements, as described in Security Features in the CRT.

Syntax

```
int vscanf_s(
   const char *format,
   va_list arglist
);
int vwscanf_s(
   const wchar_t *format,
   va_list arglist
);
```

Parameters

format
Format control string.

arglist
Variable argument list.

Return Value

Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error, or if the end-of-file character or the end-of-string character is encountered in the first attempt to read a character. If *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **vscanf_s** and **vwscanf_s** return **EOF** and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **vscanf_s** function reads data from the standard input stream **stdin** and writes the data into the locations that are given by the *arglist* argument list. Each argument in the list must be a pointer to a variable of a type that corresponds to a type specifier in *format*. If copying occurs between strings that overlap, the behavior is undefined.

vwscanf_s is a wide-character version of **vscanf_s**; the *format* argument to **vwscanf_s** is a wide-character string. **vwscanf_s** and **vscanf_s** behave identically if the stream is opened in ANSI mode. **vscanf_s** doesn't support input from a UNICODE stream.

Unlike **vscanf** and **vwscanf**, **vscanf_s** and **vwscanf_s** require the buffer size to be specified for all input parameters of type **c**, **C**, **s**, **S**, or string control sets that are enclosed in **[]**. The buffer size in characters is passed as an additional parameter immediately following the pointer to the buffer or variable. The buffer size in characters for a **wchar_t** string is not the same as the size in bytes.

The buffer size includes the terminating null. You can use a width-specification field to ensure that the token that's read in will fit into the buffer. If no width specification field is used, and the token read in is too big to fit in the

buffer, nothing is written to that buffer.

> **NOTE**
> The *size* parameter is of type **unsigned**, not **size_t**.

For more information, see scanf Width Specification.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vtscanf_s	**vscanf_s**	**vscanf_s**	**vwscanf_s**

For more information, see Format Specification Fields: scanf and wscanf Functions.

Requirements

ROUTINE	REQUIRED HEADER
vscanf_s	<stdio.h>
wscanf_s	<stdio.h> or <wchar.h>

The console is not supported in Universal Windows Platform (UWP) apps. The standard stream handles that are associated with the console, **stdin**, **stdout**, and **stderr**, must be redirected before C run-time functions can use them in UWP apps. For additional compatibility information, see Compatibility.

Example

```
// crt_vscanf_s.c
// compile with: /W3
// This program uses the vscanf_s and vwscanf_s functions
// to read formatted input.

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

int call_vscanf_s(char *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vscanf_s(format, arglist);
    va_end(arglist);
    return result;
}

int call_vwscanf_s(wchar_t *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vwscanf_s(format, arglist);
    va_end(arglist);
    return result;
}

int main( void )
{
    int   i, result;
    float fp;
    char  c, s[81];
    wchar_t wc, ws[81];
    result = call_vscanf_s("%d %f %c %C %s %S", &i, &fp, &c, 1,
                           &wc, 1, s, _countof(s), ws, _countof(ws) );
    printf( "The number of fields input is %d\n", result );
    printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c, wc, s, ws);
    result = call_vwscanf_s(L"%d %f %hc %lc %S %ls", &i, &fp, &c, 2,
                            &wc, 1, s, _countof(s), ws, _countof(ws) );
    wprintf( L"The number of fields input is %d\n", result );
    wprintf( L"The contents are: %d %f %C %c %hs %s\n", i, fp, c, wc, s, ws);
}
```

When this program is given the input in the example, it produces this output:

```
71 98.6 h z Byte characters
36 92.3 y n Wide characters
```

```
The number of fields input is 6
The contents are: 71 98.599998 h z Byte characters
The number of fields input is 6
The contents are: 36 92.300003 y n Wide characters
```

See also

Floating-Point Support
Stream I/O
Locale
printf, _printf_l, wprintf, _wprintf_l

scanf, _scanf_l, wscanf, _wscanf_l
scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l
vscanf, vwscanf

_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the number of characters in the formatted string using a pointer to a list of arguments.

Syntax

```
int _vscprintf(
   const char *format,
   va_list argptr
);
int _vscprintf_l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vscwprintf(
   const wchar_t *format,
   va_list argptr
);
int _vscwprintf_l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format-control string.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

_vscprintf returns the number of characters that would be generated if the string pointed to by the list of arguments was printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_vscwprintf** performs the same function for wide characters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Remarks

Each *argument* (if any) is converted according to the corresponding format specification in *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for printf.

> **IMPORTANT**
>
> Ensure that if *format* is a user-defined string, it is null terminated and has the correct number and type of parameters. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vsctprintf	**_vscprintf**	**_vscprintf**	**_vscwprintf**
_vsctprintf_l	**_vscprintf_l**	**_vscprintf_l**	**_vscwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER
_vscprintf, **_vscprintf_l**	<stdio.h>
_vscwprintf, **_vscwprintf_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for vsprintf.

See also

Stream I/O
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
vprintf Functions

_vscprintf_p, _vscprintf_p_l, _vscwprintf_p, _vscwprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Returns the number of characters in the formatted string using a pointer to a list of arguments, with the ability to specify the order in which the arguments are used.

Syntax

```
int _vscprintf_p(
   const char *format,
   va_list argptr
);
int _vscprintf_p _l(
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vscwprintf_p (
   const wchar_t *format,
   va_list argptr
);
int _vscwprintf_p _l(
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

format
Format-control string.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

_vscprintf_p returns the number of characters that would be generated if the string pointed to by the list of arguments was printed or sent to a file or buffer using the specified formatting codes. The value returned does not include the terminating null character. **_vscwprintf_p** performs the same function for wide characters.

Remarks

These functions differ from **_vscprintf** and **_vscwprintf** only in that they support the ability to specify the order in which the arguments are used. For more information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If *format* is a null pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

> **IMPORTANT**
>
> Ensure that if *format* is a user-defined string, it is null terminated and has the correct number and type of parameters. For more information, see Avoiding Buffer Overruns.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vsctprintf_p	**_vscprintf_p**	**_vscprintf_p**	**_vscwprintf_p**
_vsctprintf_p_l	**_vscprintf_p_l**	**_vscprintf_p_l**	**_vscwprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER
_vscprintf_p, **_vscprintf_p_l**	<stdio.h>
_vscwprintf_p, **_vscwprintf_p_l**	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

See the example for vsprintf.

See also

vprintf Functions
_scprintf_p, _scprintf_p_l, _scwprintf_p, _scwprintf_p_l
_vscprintf, _vscprintf_l, _vscwprintf, _vscwprintf_l

vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l

10/31/2018 • 5 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. More secure versions of these functions are available; see vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l.

Syntax

```
int vsnprintf(
   char *buffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf(
   char *buffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_l(
   char *buffer,
   size_t count,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vsnwprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vsnwprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsnprintf(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnprintf(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnprintf_l(
   char (&buffer)[size],
   size_t count,
   const char *format
```

```
    const char *format,
    locale_t locale,
    va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf(
    wchar_t (&buffer)[size],
    size_t count,
    const wchar_t *format,
    va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf_l(
    wchar_t (&buffer)[size],
    size_t count,
    const wchar_t *format,
    locale_t locale,
    va_list argptr
); // C++ only
```

Parameters

buffer
Storage location for output.

count
Maximum number of characters to write.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

The **vsnprintf** function returns the number of characters written, not counting the terminating null character. If the buffer size specified by *count* is not sufficiently large to contain the output specified by *format* and *argptr*, the return value of **vsnprintf** is the number of characters that would be written, not counting the null character, if *count* were sufficiently large. If the return value is greater than *count* - 1, the output has been truncated. A return value of -1 indicates that an encoding error has occurred.

Both **_vsnprintf** and **_vsnwprintf** functions return the number of characters written if the number of characters to write is less than or equal to *count*; if the number of characters to write is greater than *count*, these functions return -1 indicating that output has been truncated.

The value returned by all these functions does not include the terminating null, whether one is written or not. When *count* is zero, the value returned is the number of characters the functions would write, not including any terminating null. You can use this result to allocate sufficient buffer space for the string and its terminating null, and then call the function again to fill the buffer.

If *format* is **NULL**, or if *buffer* is **NULL** and *count* is not equal to zero, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

Remarks

Each of these functions takes a pointer to an argument list, then formats the data, and writes up to *count* characters to the memory pointed to by *buffer*. The **vsnprintf** function always writes a null terminator, even if it truncates the output. When using **_vsnprintf** and **_vsnwprintf**, the buffer will be null-terminated only if there is room at the end (that is, if the number of characters to write is less than *count*).

> **IMPORTANT**
>
> To prevent certain kinds of security risks, ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

> **NOTE**
>
> To ensure that there is room for the terminating null when calling **_vsnprintf**, **_vsnprintf_l**, **_vsnwprintf** and **_vsnwprintf_l**, be sure that *count* is strictly less than the buffer length and initialize the buffer to null prior to calling the function.
>
> Because **vsnprintf** always writes the terminating null, the *count* parameter may be equal to the size of the buffer.

Beginning with the UCRT in Visual Studio 2015 and Windows 10, **vsnprintf** is no longer identical to **_vsnprintf**. The **vsnprintf** function complies with the C99 standard; **_vnsprintf** is retained for backward compatibility with older Visual Studio code.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vsntprintf	**_vsnprintf**	**_vsnprintf**	**_vsnwprintf**
_vsntprintf_l	**_vsnprintf_l**	**_vsnprintf_l**	**_vsnwprintf_l**

Requirements

ROUTINE	REQUIRED HEADER (C)	REQUIRED HEADER (C++)
vsnprintf, **_vsnprintf**, **_vsnprintf_l**	<stdio.h>	<stdio.h> or <cstdio>
_vsnwprintf, **_vsnwprintf_l**	<stdio.h> or <wchar.h>	<stdio.h>, <wchar.h>, <cstdio>, or <cwchar>

The **_vsnprintf**, **_vsnprintf_l**, **_vsnwprintf** and **_vsnwprintf_l** functions are Microsoft specific. For additional compatibility information, see Compatibility.

Example

```
// crt_vsnwprintf.c
// compile by using: cl /W3 crt_vsnwprintf.c

// To turn off error C4996, define this symbol:
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <wtypes.h>

#define BUFFCOUNT (10)

void FormatOutput(LPCWSTR formatstring, ...)
{
    int nSize = 0;
    wchar_t buff[BUFFCOUNT];
    memset(buff, 0, sizeof(buff));
    va_list args;
    va_start(args, formatstring);
    // Note: _vsnwprintf is deprecated; consider vsnwprintf_s instead
    nSize = _vsnwprintf(buff, BUFFCOUNT - 1, formatstring, args); // C4996
    wprintf(L"nSize: %d, buff: %ls\n", nSize, buff);
    va_end(args);
}

int main() {
    FormatOutput(L"%ls %ls", L"Hi", L"there");
    FormatOutput(L"%ls %ls", L"Hi", L"there!");
    FormatOutput(L"%ls %ls", L"Hi", L"there!!");
}
```

```
nSize: 8, buff: Hi there
nSize: 9, buff: Hi there!
nSize: -1, buff: Hi there!
```

The behavior changes if you use vsnprintf instead, along with narrow-string parameters. The *count* parameter can be the entire size of the buffer, and the return value is the number of characters that would have been written if *count* was large enough:

Example

```
// crt_vsnprintf.c
// compile by using: cl /W4 crt_vsnprintf.c
#include <stdio.h>
#include <stdarg.h> // for va_list, va_start
#include <string.h> // for memset

#define BUFFCOUNT (10)

void FormatOutput(char* formatstring, ...)
{
    int nSize = 0;
    char buff[BUFFCOUNT];
    memset(buff, 0, sizeof(buff));
    va_list args;
    va_start(args, formatstring);
    nSize = vsnprintf(buff, sizeof(buff), formatstring, args);
    printf("nSize: %d, buff: %s\n", nSize, buff);
    va_end(args);
}

int main() {
    FormatOutput("%s %s", "Hi", "there");   //  8 chars + null
    FormatOutput("%s %s", "Hi", "there!");  //  9 chars + null
    FormatOutput("%s %s", "Hi", "there!!"); // 10 chars + null
}
```

```
nSize: 8, buff: Hi there
nSize: 9, buff: Hi there!
nSize: 10, buff: Hi there!
```

See also

Stream I/O
vprintf Functions
Format Specification Syntax: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vsnprintf_s, _vsnprintf_s, _vsnprintf_s_l, _vsnwprintf_s, _vsnwprintf_s_l

3/1/2019 • 3 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. These are versions of vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int vsnprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_s(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   va_list argptr
);
int _vsnprintf_s_l(
   char *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vsnwprintf_s(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vsnwprintf_s_l(
   wchar_t *buffer,
   size_t sizeOfBuffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int _vsnprintf_s(
   char (&buffer)[size],
   size_t count,
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsnwprintf_s(
   wchar_t (&buffer)[size],
   size_t count,
   const wchar_t *format,
   va_list argptr
); // C++ only
```

Parameters

buffer
Storage location for output.

sizeOfBuffer
The size of the *buffer* for output, as the character count.

count
Maximum number of characters to write (not including the terminating null), or _TRUNCATE.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

For more information, see Format Specifications.

Return Value

vsnprintf_s, **_vsnprintf_s** and **_vsnwprintf_s** return the number of characters written, not including the terminating null, or a negative value if an output error occurs. **vsnprintf_s** is identical to **_vsnprintf_s**. **vsnprintf_s** is included for compliance to the ANSI standard. **_vnsprintf** is retained for backward compatibility.

If the storage required to store the data and a terminating null exceeds *sizeOfBuffer*, the invalid parameter handler is invoked, as described in Parameter Validation, unless *count* is _TRUNCATE, in which case as much of the string as will fit in *buffer* is written and -1 returned. If execution continues after the invalid parameter handler, these functions set *buffer* to an empty string, set **errno** to **ERANGE**, and return -1.

If *buffer* or *format* is a **NULL** pointer, or if *count* is less than or equal to zero, the invalid parameter handler is invoked. If execution is allowed to continue, these functions set **errno** to **EINVAL** and return -1.

Error Conditions

CONDITION	RETURN	ERRNO
buffer is **NULL**	-1	**EINVAL**
format is **NULL**	-1	**EINVAL**
count <= 0	-1	**EINVAL**
sizeOfBuffer too small (and *count* != **_TRUNCATE**)	-1 (and *buffer* set to an empty string)	**ERANGE**

Remarks

Each of these functions takes a pointer to an argument list, then formats and writes up to *count* characters of the given data to the memory pointed to by *buffer* and appends a terminating null.

If *count* is _TRUNCATE, then these functions write as much of the string as will fit in *buffer* while leaving room for a terminating null. If the entire string (with terminating null) fits in *buffer*, then these functions return the number of characters written (not including the terminating null); otherwise, these functions return -1 to indicate that truncation occurred.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

> **NOTE**
> To ensure that there is room for the terminating null, be sure that *count* is strictly less than the buffer length, or use **_TRUNCATE**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vsntprintf_s	**_vsnprintf_s**	**_vsnprintf_s**	**_vsnwprintf_s**
_vsntprintf_s_l	**_vsnprintf_s_l**	**_vsnprintf_s_l**	**_vsnwprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vsnprintf_s	<stdio.h> and <stdarg.h>	<varargs.h>*
_vsnprintf_s, **_vsnprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
_vsnwprintf_s, **_vsnwprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt_vsnprintf_s.cpp
#include <stdio.h>
#include <wtypes.h>

void FormatOutput(LPCSTR formatstring, ...)
{
   int nSize = 0;
   char buff[10];
   memset(buff, 0, sizeof(buff));
   va_list args;
   va_start(args, formatstring);
   nSize = vsnprintf_s( buff, _countof(buff), _TRUNCATE, formatstring, args);
   printf("nSize: %d, buff: %s\n", nSize, buff);
   va_end(args);
}

int main() {
   FormatOutput("%s %s", "Hi", "there");
   FormatOutput("%s %s", "Hi", "there!");
   FormatOutput("%s %s", "Hi", "there!!");
}
```

```
nSize: 8, buff: Hi there
nSize: 9, buff: Hi there!
nSize: -1, buff: Hi there!
```

See also

Stream I/O
vprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l

3/1/2019 • 3 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. More secure versions of these functions are available; see vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l.

Syntax

```
int vsprintf(
   char *buffer,
   const char *format,
   va_list argptr
);
int _vsprintf_l(
   char *buffer,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vswprintf(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
int __vswprintf_l(
   wchar_t *buffer,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsprintf(
   char (&buffer)[size],
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vsprintf_l(
   char (&buffer)[size],
   const char *format,
   locale_t locale,
   va_list argptr
); // C++ only
template <size_t size>
int vswprintf(
   wchar_t (&buffer)[size],
   const wchar_t *format,
   va_list argptr
); // C++ only
template <size_t size>
int _vswprintf_l(
   wchar_t (&buffer)[size],
   const wchar_t *format,
   locale_t locale,
   va_list argptr
); // C++ only
```

Parameters

buffer
Storage location for output.

count
Maximum number of characters to store, in the wide string versions of this function.

format

Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

Return Value

vsprintf and **vswprintf** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *buffer* or *format* is a null pointer, these functions invoke the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by *buffer*.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

> **IMPORTANT**
> Using **vsprintf**, there is no way to limit the number of characters written, which means that code using this function is susceptible to buffer overruns. Use _vsnprintf instead, or call _vscprintf to determine how large a buffer is needed. Also, ensure that *format* is not a user-defined string. For more information, see Avoiding Buffer Overruns.

vswprintf conforms to the ISO C Standard, which requires the second parameter, *count*, of type **size_t**. To force the old nonstandard behavior, define **_CRT_NON_CONFORMING_SWPRINTFS**. The old behavior may not be in a future version, so code should be changed to use the new conformant behavior.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vstprintf	**vsprintf**	**vsprintf**	**vswprintf**
_vstprintf_l	**_vsprintf_l**	**_vsprintf_l**	**_vswprintf_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vsprintf, **_vsprintf_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vswprintf, **_vswprintf_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt_vsprintf.c
// compile with: /W3
// This program uses vsprintf to write to a buffer.
// The size of the buffer is determined by _vscprintf.

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

void test( char * format, ... )
{
    va_list args;
    int     len;
    char    *buffer;

    // retrieve the variable arguments
    va_start( args, format );

    len = _vscprintf( format, args ) // _vscprintf doesn't count
                               + 1; // terminating '\0'

    buffer = (char*)malloc( len * sizeof(char) );

    vsprintf( buffer, format, args ); // C4996
    // Note: vsprintf is deprecated; consider using vsprintf_s instead
    puts( buffer );

    free( buffer );
    va_end( args );
}

int main( void )
{
   test( "%d %c %d", 123, '<', 456 );
   test( "%s", "This is a string" );
}
```

```
123 < 456
This is a string
```

See also

Stream I/O
vprintf Functions
Format Specification Syntax: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

_vsprintf_p, _vsprintf_p_l, _vswprintf_p, _vswprintf_p_l

10/31/2018 • 2 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments, with the ability to specify the order in which the arguments are used.

Syntax

```
int _vsprintf_p(
   char *buffer,
   size_t sizeInBytes,
   const char *format,
   va_list argptr
);
int _vsprintf_p_l(
   char *buffer,
   size_t sizeInBytes,
   const char *format,
   locale_t locale,
   va_list argptr
);
int _vswprintf_p(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_p_l(
   wchar_t *buffer,
   size_t count,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
```

Parameters

buffer
Storage location for output.

sizeInBytes
Size of *buffer* in characters.

count
Maximum number of characters to store, in the UNICODE version of this function.

format
Format specification.

argptr
Pointer to list of arguments.

locale
The locale to use.

Return Value

_vsprintf_p and **_vswprintf_p** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by *buffer*.

These functions differ from the **vsprintf_s** and **vswprintf_s** only in that they support positional parameters. For more information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

If the *buffer* or *format* parameters are **NULL** pointers, if count is zero, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vstprintf_p	**_vsprintf_p**	**_vsprintf_p**	**_vswprintf_p**
_vstprintf_p_l	**_vsprintf_p_l**	**_vsprintf_p_l**	**_vswprintf_p_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
_vsprintf_p, **_vsprintf_p_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
_vswprintf_p, **_vswprintf_p_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt__vsprintf_p.c
// This program uses vsprintf_p to write to a buffer.
// The size of the buffer is determined by _vscprintf_p.

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

void example( char * format, ... )
{
    va_list  args;
    int      len;
    char     *buffer = NULL;

    va_start( args, format );

    // _vscprintf doesn't count the
    // null terminating string so we add 1.
    len = _vscprintf_p( format, args ) + 1;

    // Allocate memory for our buffer
    buffer = (char*)malloc( len * sizeof(char) );
    if (buffer)
    {
        _vsprintf_p( buffer, len, format, args );
        puts( buffer );
        free( buffer );
    }
    va_end( args );
}

int main( void )
{
    // First example
    example( "%2$d %1$c %3$d", '<', 123, 456 );

    // Second example
    example( "%s", "This is a string" );
}
```

```
123 < 456
This is a string
```

See also

Stream I/O
vprintf Functions
Format Specification Syntax: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vsprintf_s, _vsprintf_s_l, vswprintf_s, _vswprintf_s_l

3/27/2019 • 2 minutes to read • Edit Online

Write formatted output using a pointer to a list of arguments. These are versions of vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l with security enhancements as described in Security Features in the CRT.

Syntax

```
int vsprintf_s(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   va_list argptr
);
int _vsprintf_s_l(
   char *buffer,
   size_t numberOfElements,
   const char *format,
   locale_t locale,
   va_list argptr
);
int vswprintf_s(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *format,
   va_list argptr
);
int _vswprintf_s_l(
   wchar_t *buffer,
   size_t numberOfElements,
   const wchar_t *format,
   locale_t locale,
   va_list argptr
);
template <size_t size>
int vsprintf_s(
   char (&buffer)[size],
   const char *format,
   va_list argptr
); // C++ only
template <size_t size>
int vswprintf_s(
   wchar_t (&buffer)[size],
   const wchar_t *format,
   va_list argptr
); // C++ only
```

Parameters

buffer
Storage location for output.

numberOfElements
Size of *buffer* in characters.

format
Format specification.

argptr

Pointer to list of arguments.

locale
The locale to use.

Return Value

vsprintf_s and **vswprintf_s** return the number of characters written, not including the terminating null character, or a negative value if an output error occurs. If *buffer* or *format* is a null pointer, if *numberOfElements* is zero, or if the format string contains invalid formatting characters, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, the functions return -1 and set **errno** to **EINVAL**.

For information on these and other error codes, see _doserrno, errno, _sys_errlist, and _sys_nerr.

Remarks

Each of these functions takes a pointer to an argument list, and then formats and writes the given data to the memory pointed to by *buffer*.

vswprintf_s conforms to the ISO C Standard for **vswprintf**, which requires the second parameter, *count*, of type **size_t**.

These functions differ from the non-secure versions only in that the secure versions support positional parameters. For more information, see printf_p Positional Parameters.

The versions of these functions with the **_l** suffix are identical except that they use the locale parameter passed in instead of the current thread locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vstprintf_s	**vsprintf_s**	**vsprintf_s**	**vswprintf_s**
_vstprintf_s_l	**_vsprintf_s_l**	**_vsprintf_s_l**	**_vswprintf_s_l**

Requirements

ROUTINE	REQUIRED HEADER	OPTIONAL HEADERS
vsprintf_s, **_vsprintf_s_l**	<stdio.h> and <stdarg.h>	<varargs.h>*
vswprintf_s, **_vswprintf_s_l**	<stdio.h> or <wchar.h>, and <stdarg.h>	<varargs.h>*

* Required for UNIX V compatibility.

For additional compatibility information, see Compatibility.

Example

```
// crt_vsprintf_s.c
// This program uses vsprintf_s to write to a buffer.
// The size of the buffer is determined by _vscprintf.

#include <stdlib.h>
#include <stdarg.h>

void test( char * format, ... )
{
   va_list args;
   int len;
   char * buffer;

   va_start( args, format );
   len = _vscprintf( format, args ) // _vscprintf doesn't count
                               + 1; // terminating '\0'
   buffer = malloc( len * sizeof(char) );
   vsprintf_s( buffer, len, format, args );
   puts( buffer );
   free( buffer );
   va_end( args );
}

int main( void )
{
   test( "%d %c %d", 123, '<', 456 );
   test( "%s", "This is a string" );
}
```

```
123 < 456
This is a string
```

See also

Stream I/O
vprintf Functions
Format Specification Syntax: printf and wprintf Functions
fprintf, _fprintf_l, fwprintf, _fwprintf_l
printf, _printf_l, wprintf, _wprintf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
va_arg, va_copy, va_end, va_start

vsscanf, vswscanf

10/31/2018 • 2 minutes to read • Edit Online

Reads formatted data from a string. More secure versions of these functions are available; see vsscanf_s, vswscanf_s.

Syntax

```
int vsscanf(
   const char *buffer,
   const char *format,
   va_list arglist
);
int vswscanf(
   const wchar_t *buffer,
   const wchar_t *format,
   va_list arglist
);
```

Parameters

buffer
Stored data

format
Format-control string. For more information, see Format Specification Fields: scanf and wscanf Functions.

arglist
Variable argument list.

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **vsscanf** function reads data from *buffer* into the locations that are given by each argument in the *arglist* argument list. Every argument in the list must be a pointer to a variable that has a type that corresponds to a type specifier in *format*. The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf** function. If copying takes place between strings that overlap, the behavior is undefined.

> **IMPORTANT**
>
> When you use **vsscanf** to read a string, always specify a width for the **%s** format (for example, **"%32s"** instead of **"%s"**); otherwise, incorrectly formatted input can cause a buffer overrun.

vswscanf is a wide-character version of **vsscanf**; the arguments to **vswscanf** are wide-character strings. **vsscanf** does not handle multibyte hexadecimal characters. **vswscanf** does not handle Unicode full-width hexadecimal or "compatibility zone" characters. Otherwise, **vswscanf** and **vsscanf** behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vstscanf	**vsscanf**	**vsscanf**	**vswscanf**

Requirements

ROUTINE	REQUIRED HEADER
vsscanf	<stdio.h>
vswscanf	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_vsscanf.c
// compile with: /W3
// This program uses vsscanf to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>
#include <stdarg.h>

int call_vsscanf(char *tokenstring, char *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vsscanf(tokenstring, format, arglist);
    va_end(arglist);
    return result;
}

int main( void )
{
    char  tokenstring[] = "15 12 14...";
    char  s[81];
    char  c;
    int   i;
    float fp;

    // Input various data from tokenstring:
    // max 80 character string:
    call_vsscanf(tokenstring, "%80s", s);
    call_vsscanf(tokenstring, "%c", &c);
    call_vsscanf(tokenstring, "%d", &i);
    call_vsscanf(tokenstring, "%f", &fp);

    // Output the data read
    printf("String    = %s\n", s);
    printf("Character = %c\n", c);
    printf("Integer:  = %d\n", i);
    printf("Real:     = %f\n", fp);
}
```

```
String    = 15
Character = 1
Integer:  = 15
Real:     = 15.000000
```

See also

Stream I/O
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vsscanf_s, vswscanf_s

vsscanf_s, vswscanf_s

10/31/2018 • 3 minutes to read • Edit Online

Reads formatted data from a string. These versions of vsscanf, vswscanf have security enhancements, as described in Security Features in the CRT.

Syntax

```
int vsscanf_s(
   const char *buffer,
   const char *format,
   va_list argptr
);
int vswscanf_s(
   const wchar_t *buffer,
   const wchar_t *format,
   va_list arglist
);
```

Parameters

buffer
Stored data

format
Format-control string. For more information, see Format Specification Fields: scanf and wscanf Functions.

arglist
Variable argument list.

Return Value

Each of these functions returns the number of fields that are successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** for an error or if the end of the string is reached before the first conversion.

If *buffer* or *format* is a **NULL** pointer, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, these functions return -1 and set **errno** to **EINVAL**.

For information about these and other error codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

Remarks

The **vsscanf_s** function reads data from *buffer* into the locations that are given by each argument in the *arglist* argument list. The arguments in the argument list specify pointers to variables that have a type that corresponds to a type specifier in *format*. Unlike the less secure version **vsscanf**, a buffer size parameter is required when you use the type field characters **c**, **C**, **s**, **S**, or string-control sets that are enclosed in **[]**. The buffer size in characters must be supplied as an additional parameter immediately after each buffer parameter that requires it.

The buffer size includes the terminating null. A width specification field may be used to ensure that the token that's read in will fit into the buffer. If no width specification field is used, and the token read in is too big to fit in the buffer, nothing is written to that buffer.

For more information, see scanf_s, _scanf_s_l, wscanf_s, _wscanf_s_l and scanf Type Field Characters.

> **NOTE**
> The size parameter is of type **unsigned**, not **size_t**.

The *format* argument controls the interpretation of the input fields and has the same form and function as the *format* argument for the **scanf_s** function. If copying occurs between strings that overlap, the behavior is undefined.

vswscanf_s is a wide-character version of **vsscanf_s**; the arguments to **vswscanf_s** are wide-character strings. **vsscanf_s** does not handle multibyte hexadecimal characters. **vswscanf_s** does not handle Unicode full-width hexadecimal or "compatibility zone" characters. Otherwise, **vswscanf_s** and **vsscanf_s** behave identically.

Generic-Text Routine Mappings

TCHAR.H ROUTINE	_UNICODE & _MBCS NOT DEFINED	_MBCS DEFINED	_UNICODE DEFINED
_vstscanf_s	**vsscanf_s**	**vsscanf_s**	**vswscanf_s**

Requirements

ROUTINE	REQUIRED HEADER
vsscanf_s	<stdio.h>
vswscanf_s	<stdio.h> or <wchar.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_vsscanf_s.c
// compile with: /W3
// This program uses vsscanf_s to read data items
// from a string named tokenstring, then displays them.

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

int call_vsscanf_s(char *tokenstring, char *format, ...)
{
    int result;
    va_list arglist;
    va_start(arglist, format);
    result = vsscanf_s(tokenstring, format, arglist);
    va_end(arglist);
    return result;
}

int main( void )
{
    char  tokenstring[] = "15 12 14...";
    char  s[81];
    char  c;
    int   i;
    float fp;

    // Input various data from tokenstring:
    // max 80 character string:
    call_vsscanf_s(tokenstring, "%80s", s, _countof(s));
    call_vsscanf_s(tokenstring, "%c", &c, sizeof(char));
    call_vsscanf_s(tokenstring, "%d", &i);
    call_vsscanf_s(tokenstring, "%f", &fp);

    // Output the data read
    printf("String    = %s\n", s);
    printf("Character = %c\n", c);
    printf("Integer:  = %d\n", i);
    printf("Real:     = %f\n", fp);
}
```

```
String    = 15
Character = 1
Integer:  = 15
Real:     = 15.000000
```

See also

Stream I/O
scanf, _scanf_l, wscanf, _wscanf_l
sscanf, _sscanf_l, swscanf, _swscanf_l
sscanf_s, _sscanf_s_l, swscanf_s, _swscanf_s_l
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vsscanf, vswscanf

wcrtomb

10/31/2018 • 2 minutes to read • Edit Online

Convert a wide character into its multibyte character representation. A more secure version of this function is available; see wcrtomb_s.

Syntax

```
size_t wcrtomb(
   char *mbchar,
   wchar_t wchar,
   mbstate_t *mbstate
);
template <size_t size>
size_t wcrtomb(
   char (&mbchar)[size],
   wchar_t wchar,
   mbstate_t *mbstate
); // C++ only
```

Parameters

mbchar
The resulting multibyte converted character.

wchar
A wide character to convert.

mbstate
A pointer to an **mbstate_t** object.

Return Value

Returns the number of bytes required to represent the converted multibyte character, otherwise a -1 if an error occurs.

Remarks

The **wcrtomb** function converts a wide character, beginning in the specified conversion state contained in *mbstate*, from the value contained in *wchar*, into the address represented by *mbchar*. The return value is the number of bytes required to represent the corresponding multibyte character, but it will not return more than **MB_CUR_MAX** bytes.

If *mbstate* is null, the internal **mbstate_t** object containing the conversion state of *mbchar* is used. If the character sequence *wchar* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

The **wcrtomb** function differs from wctomb, _wctomb_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcsnlen**, if a subsequent call to **wcsrtombs** were used instead of **wcstombs**.

In C++, this function has a template overload that invokes the newer, secure counterparts of this function. For

more information, see Secure Template Overloads.

Exceptions

The **wcrtomb** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and while the *mbstate* is null.

Example

```
// crt_wcrtomb.c
// compile with: /W3
// This program converts a wide character
// to its corresponding multibyte character.

#include <string.h>
#include <stdio.h>
#include <wchar.h>

int main( void )
{
    size_t      sizeOfCovertion = 0;
    mbstate_t   mbstate;
    char        mbStr = 0;
    wchar_t*    wcStr = L"Q";

    // Reset to initial conversion state
    memset(&mbstate, 0, sizeof(mbstate));

    sizeOfCovertion = wcrtomb(&mbStr, *wcStr, &mbstate); // C4996
    // Note: wcrtomb is deprecated; consider using wcrtomb_s instead
    if (sizeOfCovertion > 0)
    {
        printf("The corresponding wide character \"");
        wprintf(L"%s\"", wcStr);
        printf(" was converted to the \"%c\" ", mbStr);
        printf("multibyte character.\n");
    }
    else
    {
        printf("No corresponding multibyte character "
               "was found.\n");
    }
}
```

```
The corresponding wide character "Q" was converted to the "Q" multibyte character.
```

Requirements

ROUTINE	REQUIRED HEADER
wcrtomb	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbsinit

wcrtomb_s

10/31/2018 • 2 minutes to read • Edit Online

Convert a wide character into its multibyte character representation. A version of wcrtomb with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t wcrtomb_s(
   size_t *pReturnValue,
   char *mbchar,
   size_t sizeOfmbchar,
   wchar_t *wchar,
   mbstate_t *mbstate
);
template <size_t size>
errno_t wcrtomb_s(
   size_t *pReturnValue,
   char (&mbchar)[size],
   wchar_t *wchar,
   mbstate_t *mbstate
); // C++ only
```

Parameters

pReturnValue
Returns the number of bytes written or -1 if an error occurred.

mbchar
The resulting multibyte converted character.

sizeOfmbchar
The size of the *mbchar* variable in bytes.

wchar
A wide character to convert.

mbstate
A pointer to an **mbstate_t** object.

Return Value

Returns zero or an **errno** value if an error occurs.

Remarks

The **wcrtomb_s** function converts a wide character, beginning in the specified conversion state contained in *mbstate*, from the value contained in *wchar*, into the address represented by *mbchar*. The *pReturnValue* value will be the number of bytes converted, but no more than **MB_CUR_MAX** bytes, or an -1 if an error occurred.

If *mbstate* is null, the internal **mbstate_t** conversion state is used. If the character contained in *wchar* does not have a corresponding multibyte character, the value of *pReturnValue* will be -1 and the function will return the **errno** value of **EILSEQ**.

The **wcrtomb_s** function differs from wctomb_s, _wctomb_s_l by its restartability. The conversion state is stored

in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs_s** were used instead of **wcstombs_s**.

In C++, using this function is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **wcrtomb_s** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

Example

```
// crt_wcrtomb_s.c
// This program converts a wide character
// to its corresponding multibyte character.
//

#include <string.h>
#include <stdio.h>
#include <wchar.h>

int main( void )
{
    errno_t     returnValue;
    size_t      pReturnValue;
    mbstate_t   mbstate;
    size_t      sizeOfmbStr = 1;
    char        mbchar = 0;
    wchar_t*    wchar = L"Q\0";

    // Reset to initial conversion state
    memset(&mbstate, 0, sizeof(mbstate));

    returnValue = wcrtomb_s(&pReturnValue, &mbchar, sizeof(char),
                            *wchar, &mbstate);
    if (returnValue == 0) {
        printf("The corresponding wide character \"");
        wprintf(L"%s\"", wchar);
        printf(" was converted to a the \"%c\" ", mbchar);
        printf("multibyte character.\n");
    }
    else
    {
        printf("No corresponding multibyte character "
               "was found.\n");
    }
}
```

```
The corresponding wide character "Q" was converted to a the "Q" multibyte character.
```

Requirements

ROUTINE	REQUIRED HEADER
wcrtomb_s	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
mbsinit

wcsrtombs

10/31/2018 • 2 minutes to read • Edit Online

Convert a wide character string to its multibyte character string representation. A more secure version of this function is available; see wcsrtombs_s.

Syntax

```
size_t wcsrtombs(
   char *mbstr,
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
size_t wcsrtombs(
   char (&mbstr)[size],
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

mbstr
The resulting converted multibyte character string's address location.

wcstr
Indirectly points to the location of the wide character string to be converted.

count
The number of character to be converted.

mbstate
A pointer to an **mbstate_t** conversion state object.

Return Value

Returns the number of bytes successfully converted, not including the null terminating null byte (if any), otherwise a -1 if an error occurred.

Remarks

The **wcsrtombs** function converts a string of wide characters, beginning in the specified conversion state contained in *mbstate*, from the values indirect pointed to in *wcstr*, into the address of *mbstr*. The conversion will continue for each character until: after a null terminating wide character is encountered, when a non corresponding character is encountered or when the next character would exceed the limit contained in *count*. If **wcsrtombs** encounters the wide-character null character (L'\0') either before or when *count* occurs, it converts it to an 8-bit 0 and stops.

Thus, the multibyte character string at *mbstr* is null-terminated only if **wcsrtombs** encounters a wide character null character during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcsrtombs** is undefined. **wcsrtombs** is affected by the LC_TYPE category of the current locale.

The **wcsrtombs** function differs from wcstombs, _wcstombs_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcsnlen**, if a subsequent call to **wcsrtombs** were used instead of **wcstombs**.

If the *mbstr* argument is **NULL**, **wcsrtombs** returns the required size in bytes of the destination string. If *mbstate* is null, the internal **mbstate_t** conversion state is used. If the character sequence *wchar* does not have a corresponding multibyte character representation, a -1 is returned and the **errno** is set to **EILSEQ**.

In C++, this function has a template overload that invokes the newer, secure counterpart of this function. For more information, see Secure Template Overloads.

Exceptions

The **wcsrtombs** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is not null.

Example

```
// crt_wcsrtombs.cpp
// compile with: /W3
// This code example converts a wide
// character string into a multibyte
// character string.

#include <stdio.h>
#include <memory.h>
#include <wchar.h>
#include <errno.h>

#define MB_BUFFER_SIZE 100

int main()
{
    const wchar_t   wcString[] =
                    {L"Every good boy does fine."};
    const wchar_t   *wcsIndirectString = wcString;
    char            mbString[MB_BUFFER_SIZE];
    size_t          countConverted;
    mbstate_t       mbstate;

    // Reset to initial shift state
    ::memset((void*)&mbstate, 0, sizeof(mbstate));

    countConverted = wcsrtombs(mbString, &wcsIndirectString,
                               MB_BUFFER_SIZE, &mbstate); // C4996
    // Note: wcsrtombs is deprecated; consider using wcsrtombs_s
    if (errno == EILSEQ)
    {
        printf( "An encoding error was detected in the string.\n" );
    }
    else
    {
        printf( "The string was successfuly converted.\n" );
    }
}
```

```
The string was successfuly converted.
```

Requirements

ROUTINE	REQUIRED HEADER
wcsrtombs	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
wcrtomb
wcrtomb_s
wctomb, _wctomb_l
wcstombs, _wcstombs_l
mbsinit

wcsrtombs_s

4/22/2019 • 3 minutes to read • Edit Online

Convert a wide character string to its multibyte character string representation. A version of wcsrtombs with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t wcsrtombs_s(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
);
template <size_t size>
errno_t wcsrtombs_s(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t **wcstr,
   sizeof count,
   mbstate_t *mbstate
); // C++ only
```

Parameters

pReturnValue
The size in bytes of the converted string, including the null terminator.

mbstr
The address of a buffer for the resulting converted multibyte character string.

sizeInBytes
The size in bytes of the *mbstr* buffer.

wcstr
Points to the wide character string to be converted.

count
The maximum number of bytes to be stored in the *mbstr* buffer, or _TRUNCATE.

mbstate
A pointer to an **mbstate_t** conversion state object.

Return Value

Zero if successful, an error code on failure.

ERROR CONDITION	RETURN VALUE AND ERRNO
mbstr is **NULL** and *sizeInBytes* > 0	**EINVAL**
wcstr is **NULL**	**EINVAL**

ERROR CONDITION	RETURN VALUE AND ERRNO
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation . If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **wcsrtombs_s** function converts a string of wide characters pointed to by *wcstr* into multibyte characters stored in the buffer pointed to by *mbstr*, using the conversion state contained in *mbstate*. The conversion will continue for each character until one of these conditions is met:

- A null wide character is encountered
- A wide character that cannot be converted is encountered
- The number of bytes stored in the *mbstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **wcsrtombs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator.

If **wcsrtombs_s** successfully converts the source string, it puts the size in bytes of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *mbstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *mbstr* is **NULL**, *count* is ignored.

If **wcsrtombs_s** encounters a wide character it cannot convert to a multibyte character, it puts -1 in **pReturnValue*, sets the destination buffer to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcsrtombs_s** is undefined. **wcsrtombs_s** is affected by the LC_TYPE category of the current locale.

> **IMPORTANT**
> Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of wide characters to convert.

The **wcsrtombs_s** function differs from wcstombs_s, _wcstombs_s_l by its restartability. The conversion state is stored in *mbstate* for subsequent calls to the same or other restartable functions. Results are undefined when mixing the use of restartable and nonrestartable functions. For example, an application would use **wcsrlen** rather than **wcslen**, if a subsequent call to **wcsrtombs_s** were used instead of **wcstombs_s**.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Exceptions

The **wcsrtombs_s** function is multithread safe as long as no function in the current thread calls **setlocale** while this function is executing and the *mbstate* is null.

Example

```
// crt_wcsrtombs_s.cpp
//
// This code example converts a wide
// character string into a multibyte
// character string.
//

#include <stdio.h>
#include <memory.h>
#include <wchar.h>
#include <errno.h>

#define MB_BUFFER_SIZE 100

void main()
{
    const wchar_t   wcString[] =
                    {L"Every good boy does fine."};
    const wchar_t   *wcsIndirectString = wcString;
    char            mbString[MB_BUFFER_SIZE];
    size_t          countConverted;
    errno_t         err;
    mbstate_t       mbstate;

    // Reset to initial shift state
    ::memset((void*)&mbstate, 0, sizeof(mbstate));

    err = wcsrtombs_s(&countConverted, mbString, MB_BUFFER_SIZE,
                      &wcsIndirectString, MB_BUFFER_SIZE, &mbstate);
    if (err == EILSEQ)
    {
        printf( "An encoding error was detected in the string.\n" );
    }
    else
    {
        printf( "The string was successfully converted.\n" );
    }
}
```

```
The string was successfully converted.
```

Requirements

ROUTINE	REQUIRED HEADER
wcsrtombs_s	<wchar.h>

See also

Data Conversion
Locale
Interpretation of Multibyte-Character Sequences
wcrtomb
wcrtomb_s
wctomb, _wctomb_l
wcstombs, _wcstombs_l
mbsinit

wcstombs, _wcstombs_l

3/1/2019 • 2 minutes to read • Edit Online

Converts a sequence of wide characters to a corresponding sequence of multibyte characters. More secure versions of these functions are available; see wcstombs_s, _wcstombs_s_l.

Syntax

```
size_t wcstombs(
   char *mbstr,
   const wchar_t *wcstr,
   size_t count
);
size_t _wcstombs_l(
   char *mbstr,
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
);
template <size_t size>
size_t wcstombs(
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count
); // C++ only
template <size_t size>
size_t _wcstombs_l(
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

mbstr
The address of a sequence of multibyte characters.

wcstr
The address of a sequence of wide characters.

count
The maximum number of bytes that can be stored in the multibyte output string.

locale
The locale to use.

Return Value

If **wcstombs** successfully converts the multibyte string, it returns the number of bytes written into the multibyte output string, excluding the terminating null (if any). If the *mbstr* argument is **NULL**, **wcstombs** returns the required size in bytes of the destination string. If **wcstombs** encounters a wide character it cannot convert to a multibyte character, it returns -1 cast to type **size_t** and sets **errno** to **EILSEQ**.

Remarks

The **wcstombs** function converts the wide-character string pointed to by *wcstr* to the corresponding multibyte characters and stores the results in the *mbstr* array. The *count* parameter indicates the maximum number of bytes that can be stored in the multibyte output string (that is, the size of *mbstr*). In general, it is not known how many bytes will be required when converting a wide-character string. Some wide characters will require only one byte in the output string; others require two. If there are two bytes in the multibyte output string for every wide character in the input string (including the wide character null), the result is guaranteed to fit.

If **wcstombs** encounters the wide-character null character (L'\0') either before or when *count* occurs, it converts it to an 8-bit 0 and stops. Thus, the multibyte character string at *mbstr* is null-terminated only if **wcstombs** encounters a wide-character null character during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcstombs** is undefined.

If the *mbstr* argument is **NULL**, **wcstombs** returns the required size in bytes of the destination string.

wcstombs validates its parameters. If *wcstr* is **NULL**, or if *count* is greater than **INT_MAX**, this function invokes the invalid parameter handler, as described in Parameter Validation . If execution is allowed to continue, the function sets **errno** to **EINVAL** and returns -1.

wcstombs uses the current locale for any locale-dependent behavior; **_wcstombs_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

In C++, these functions have template overloads that invoke the newer, secure counterparts of these functions. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
wcstombs	<stdlib.h>
_wcstombs_l	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

This program illustrates the behavior of the **wcstombs** function.

```
// crt_wcstombs.c
// compile with: /W3
// This example demonstrates the use
// of wcstombs, which converts a string
// of wide characters to a string of
// multibyte characters.

#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE 100

int main( void )
{
    size_t  count;
    char    *pMBBuffer = (char *)malloc( BUFFER_SIZE );
    wchar_t *pWCBuffer = L"Hello, world.";

    printf("Convert wide-character string:\n" );

    count = wcstombs(pMBBuffer, pWCBuffer, BUFFER_SIZE ); // C4996
    // Note: wcstombs is deprecated; consider using wcstombs_s instead
    printf("   Characters converted: %u\n",
            count );
    printf("    Multibyte character: %s\n\n",
           pMBBuffer );

    free(pMBBuffer);
}
```

```
Convert wide-character string:
   Characters converted: 13
    Multibyte character: Hello, world.
```

See also

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wctomb, _wctomb_l
WideCharToMultiByte

wcstombs_s, _wcstombs_s_l

4/22/2019 • 3 minutes to read • Edit Online

Converts a sequence of wide characters to a corresponding sequence of multibyte characters. A version of wcstombs, _wcstombs_l with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t wcstombs_s(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t *wcstr,
   size_t count
);

errno_t _wcstombs_s_l(
   size_t *pReturnValue,
   char *mbstr,
   size_t sizeInBytes,
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
);

template <size_t size>
errno_t wcstombs_s(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count
); // C++ only

template <size_t size>
errno_t _wcstombs_s_l(
   size_t *pReturnValue,
   char (&mbstr)[size],
   const wchar_t *wcstr,
   size_t count,
   _locale_t locale
); // C++ only
```

Parameters

pReturnValue
The size in bytes of the converted string, including the null terminator.

mbstr
The address of a buffer for the resulting converted multibyte character string.

sizeInBytes
The size in bytes of the *mbstr* buffer.

wcstr
Points to the wide character string to be converted.

count
The maximum number of bytes to store in the *mbstr* buffer, not including the terminating null character, or

_TRUNCATE.

locale
The locale to use.

Return Value

Zero if successful, an error code on failure.

ERROR CONDITION	RETURN VALUE AND ERRNO
mbstr is **NULL** and *sizeInBytes* > 0	**EINVAL**
wcstr is **NULL**	**EINVAL**
The destination buffer is too small to contain the converted string (unless *count* is **_TRUNCATE**; see Remarks below)	**ERANGE**

If any of these conditions occurs, the invalid parameter exception is invoked as described in Parameter Validation. If execution is allowed to continue, the function returns an error code and sets **errno** as indicated in the table.

Remarks

The **wcstombs_s** function converts a string of wide characters pointed to by *wcstr* into multibyte characters stored in the buffer pointed to by *mbstr*. The conversion will continue for each character until one of these conditions is met:

- A null wide character is encountered
- A wide character that cannot be converted is encountered
- The number of bytes stored in the *mbstr* buffer equals *count*.

The destination string is always null-terminated (even in the case of an error).

If *count* is the special value _TRUNCATE, then **wcstombs_s** converts as much of the string as will fit into the destination buffer, while still leaving room for a null terminator. If the string is truncated, the return value is **STRUNCATE**, and the conversion is considered successful.

If **wcstombs_s** successfully converts the source string, it puts the size in bytes of the converted string, including the null terminator, into **pReturnValue* (provided *pReturnValue* is not **NULL**). This occurs even if the *mbstr* argument is **NULL** and provides a way to determine the required buffer size. Note that if *mbstr* is **NULL**, *count* is ignored.

If **wcstombs_s** encounters a wide character it cannot convert to a multibyte character, it puts 0 in **pReturnValue*, sets the destination buffer to an empty string, sets **errno** to **EILSEQ**, and returns **EILSEQ**.

If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of **wcstombs_s** is undefined.

> **IMPORTANT**
> Ensure that *wcstr* and *mbstr* do not overlap, and that *count* correctly reflects the number of wide characters to convert.

wcstombs_s uses the current locale for any locale-dependent behavior; **_wcstombs_s_l** is identical to **wcstombs** except that it uses the locale passed in instead. For more information, see Locale.

In C++, using these functions is simplified by template overloads; the overloads can infer buffer length

automatically (eliminating the need to specify a size argument) and they can automatically replace older, non-secure functions with their newer, secure counterparts. For more information, see Secure Template Overloads.

Requirements

ROUTINE	REQUIRED HEADER
wcstombs_s	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

This program illustrates the behavior of the **wcstombs_s** function.

```
// crt_wcstombs_s.c
// This example converts a wide character
// string to a multibyte character string.
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BUFFER_SIZE 100

int main( void )
{
    size_t   i;
    char      *pMBBuffer = (char *)malloc( BUFFER_SIZE );
    wchar_t*pWCBuffer = L"Hello, world.";

    printf( "Convert wide-character string:\n" );

    // Conversion
    wcstombs_s(&i, pMBBuffer, (size_t)BUFFER_SIZE,
               pWCBuffer, (size_t)BUFFER_SIZE );

    // Output
    printf("   Characters converted: %u\n", i);
    printf("    Multibyte character: %s\n\n",
     pMBBuffer );

    // Free multibyte character buffer
    if (pMBBuffer)
    {
    free(pMBBuffer);
    }
}
```

```
Convert wide-character string:
   Characters converted: 14
    Multibyte character: Hello, world.
```

See also

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l

wctomb_s, _wctomb_s_l
WideCharToMultiByte

wctob

11/9/2018 • 2 minutes to read • Edit Online

Determines if a wide character corresponds to a multibyte character and returns its multibyte character representation.

Syntax

```
int wctob(
   wint_t wchar
);
```

Parameters

wchar
Value to translate.

Return Value

If **wctob** successfully converts a wide character, it returns its multibyte character representation, only if the multibyte character is exactly one byte long. If **wctob** encounters a wide character it cannot convert to a multibyte character or the multibyte character is not exactly one byte long, it returns a -1.

Remarks

The **wctob** function converts a wide character contained in *wchar* to the corresponding multibyte character passed by the return **int** value, if the multibyte character is exactly one byte long.

If **wctob** was unsuccessful and no corresponding multibyte character was found, the function sets **errno** to **EILSEQ** and returns -1.

Requirements

ROUTINE	REQUIRED HEADER
wctob	<wchar.h>

For additional compatibility information, see Compatibility.

Example

This program illustrates the behavior of the **wcstombs** function.

```
// crt_wctob.c
#include <stdio.h>
#include <wchar.h>

int main( void )
{
    int     bChar = 0;
    wint_t  wChar = 0;

    // Set the corresponding wide character to exactly one byte.
    wChar = (wint_t)'A';

    bChar = wctob( wChar );
    if (bChar == WEOF)
    {
        printf( "No corresponding multibyte character was found.\n");
    }
    else
    {
        printf( "Determined the corresponding multibyte character to"
                " be \"%c\".\n", bChar);
    }
}
```

```
Determined the corresponding multibyte character to be "A".
```

See also

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wctomb, _wctomb_l
WideCharToMultiByte

wctomb, _wctomb_l

3/1/2019 • 2 minutes to read • Edit Online

Convert a wide character to the corresponding multibyte character. More secure versions of these functions are available; see wctomb_s, _wctomb_s_l.

Syntax

```
int wctomb(
   char *mbchar,
   wchar_t wchar
);
int _wctomb_l(
   char *mbchar,
   wchar_t wchar,
   _locale_t locale
);
```

Parameters

mbchar
The address of a multibyte character.

wchar
A wide character.

Return Value

If **wctomb** converts the wide character to a multibyte character, it returns the number of bytes (which is never greater than **MB_CUR_MAX**) in the wide character. If *wchar* is the wide-character null character (L'\0'), **wctomb** returns 1. If the target pointer *mbchar* is **NULL**, **wctomb** returns 0. If the conversion is not possible in the current locale, **wctomb** returns -1 and **errno** is set to **EILSEQ**.

Remarks

The **wctomb** function converts its *wchar* argument to the corresponding multibyte character and stores the result at *mbchar*. You can call the function from any point in any program. **wctomb** uses the current locale for any locale-dependent behavior; **_wctomb_l** is identical to **wctomb** except that it uses the locale passed in instead. For more information, see Locale.

wctomb validates its parameters. If *mbchar* is **NULL**, the invalid parameter handler is invoked, as described in Parameter Validation. If execution is allowed to continue, **errno** is set to **EINVAL** and the function returns -1.

Requirements

ROUTINE	REQUIRED HEADER
wctomb	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

This program illustrates the behavior of the wctomb function.

```
// crt_wctomb.cpp
// compile with: /W3
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   int i;
   wchar_t wc = L'a';
   char *pmb = (char *)malloc( MB_CUR_MAX );

   printf( "Convert a wide character:\n" );
   i = wctomb( pmb, wc ); // C4996
   // Note: wctomb is deprecated; consider using wctomb_s
   printf( "   Characters converted: %u\n", i );
   printf( "   Multibyte character: %.1s\n\n", pmb );
}
```

```
Convert a wide character:
   Characters converted: 1
   Multibyte character: a
```

See also

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l
mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
WideCharToMultiByte

wctomb_s, _wctomb_s_l

10/31/2018 • 2 minutes to read • Edit Online

Converts a wide character to the corresponding multibyte character. A version of wctomb, _wctomb_l with security enhancements as described in Security Features in the CRT.

Syntax

```
errno_t wctomb_s(
   int *pRetValue,
   char *mbchar,
   size_t sizeInBytes,
   wchar_t wchar
);
errno_t _wctomb_s_l(
   int *pRetValue,
   char *mbchar,
   size_t sizeInBytes,
   wchar_t wchar,
   _locale_t locale
);
```

Parameters

pRetValue
The number of bytes, or a code indicating the result.

mbchar
The address of a multibyte character.

sizeInBytes
Size of the buffer *mbchar*.

wchar
A wide character.

locale
The locale to use.

Return Value

Zero if successful, an error code on failure.

Error Conditions

MBCHAR	*SIZEINBYTES*	RETURN VALUE	*PRETVALUE*
NULL	>0	**EINVAL**	not modified
any	>**INT_MAX**	**EINVAL**	not modified
any	too small	**EINVAL**	not modified

If any of the above error conditions occurs, the invalid parameter handler is invoked, as described in Parameter

Validation. If execution is allowed to continue, **wctomb** returns **EINVAL** and sets **errno** to **EINVAL**.

Remarks

The **wctomb_s** function converts its *wchar* argument to the corresponding multibyte character and stores the result at *mbchar*. You can call the function from any point in any program.

If **wctomb_s** converts the wide character to a multibyte character, it puts the number of bytes (which is never greater than **MB_CUR_MAX**) in the wide character into the integer pointed to by *pRetValue*. If *wchar* is the wide-character null character (L'\0'), **wctomb_s** fills *pRetValue* with 1. If the target pointer *mbchar* is **NULL**, **wctomb_s** puts 0 in *pRetValue*. If the conversion is not possible in the current locale, **wctomb_s** puts -1 in *pRetValue*.

wctomb_s uses the current locale for locale-dependent information; **_wctomb_s_l** is identical except that it uses the locale passed in instead. For more information, see Locale.

Requirements

ROUTINE	REQUIRED HEADER
wctomb_s	<stdlib.h>
_wctomb_s_l	<stdlib.h>

For additional compatibility information, see Compatibility.

Example

This program illustrates the behavior of the **wctomb** function.

```
// crt_wctomb_s.cpp
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int i;
    wchar_t wc = L'a';
    char *pmb = (char *)malloc( MB_CUR_MAX );

    printf_s( "Convert a wide character:\n" );
    wctomb_s( &i, pmb, MB_CUR_MAX, wc );
    printf_s( "   Characters converted: %u\n", i );
    printf_s( "   Multibyte character: %.1s\n\n", pmb );
}
```

```
Convert a wide character:
   Characters converted: 1
   Multibyte character: a
```

See also

Data Conversion
Locale
_mbclen, mblen, _mblen_l
mbstowcs, _mbstowcs_l

mbtowc, _mbtowc_l
wcstombs, _wcstombs_l
WideCharToMultiByte

wctrans

10/31/2018 • 2 minutes to read • Edit Online

Determines a mapping from one set of character codes to another.

Syntax

```
wctrans_t wctrans(
   const char *property
);
```

Parameters

property
A string that specifies one of the valid transformations.

Return Value

If the **LC_CTYPE** category of the current locale does not define a mapping whose name matches the property string *property*, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to towctrans.

Remarks

This function determines a mapping from one set of character codes to another.

The following pairs of calls have the same behavior in all locales, but it is possible to define additional mappings even in the "C" locale:

FUNCTION	SAME AS
tolower(c)	towctrans(c, wctrans("towlower"))
towupper(c)	towctrans(c, wctrans("toupper"))

Requirements

ROUTINE	REQUIRED HEADER
wctrans	<wctype.h>

For additional compatibility information, see Compatibility.

Example

```
// crt_wctrans.cpp
// compile with: /EHsc
// This example determines a mapping from one set of character
// codes to another.

#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <iostream>

int main()
{
    wint_t c = 'a';
    printf_s("%d\n",c);

    wctrans_t i = wctrans("toupper");
    printf_s("%d\n",i);

    wctrans_t ii = wctrans("towlower");
    printf_s("%d\n",ii);

    wchar_t wc = towctrans(c, i);
    printf_s("%d\n",wc);
}
```

```
97
1
0
65
```

See also

Data Conversion

setlocale, _wsetlocale

wctype

10/31/2018 • 2 minutes to read • Edit Online

Determines a classification rule for wide-character codes.

Syntax

```
wctype_t wctype(
   const char * property
);
```

Parameters

property
Property string.

Return Value

If the **LC_CTYPE** category of the current locale does not define a classification rule whose name matches the property string *property*, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to towctrans.

Remarks

The function determines a classification rule for wide-character codes. The following pairs of calls have the same behavior in all locales (but an implementation can define additional classification rules even in the "C" locale):

FUNCTION	SAME AS
iswalnum(c)	iswctype(c, wctype("alnum"))
iswalpha(c)	iswctype(c, wctype("alpha"))
iswcntrl(c)	iswctype(c, wctype("cntrl"))
iswdigit(c)	iswctype(c, wctype("digit"))
iswgraph(c)	iswctype(c, wctype("graph"))
iswlower(c)	iswctype(c, wctype("lower"))
iswprint(c)	iswctype(c, wctype("print"))
iswpunct(c)	iswctype(c, wctype("punct"))
iswspace(c)	iswctype(c, wctype("space"))
iswupper(c)	iswctype(c, wctype("upper"))

FUNCTION	SAME AS
iswxdigit(c)	iswctype(c, wctype("xdigit"))

Requirements

ROUTINE	REQUIRED HEADER
wctype	<wctype.h>

For additional compatibility information, see Compatibility.

See also

Data Conversion
setlocale, _wsetlocale

write

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _write instead.

_write

10/31/2018 • 2 minutes to read • Edit Online

Writes data to a file.

Syntax

```
int _write(
   int fd,
   const void *buffer,
   unsigned int count
);
```

Parameters

fd
File descriptor of file into which data is written.

buffer
Data to be written.

count
Number of bytes.

Return Value

If successful, **_write** returns the number of bytes actually written. If the actual space remaining on the disk is less than the size of the buffer the function is trying to write to the disk, **_write** fails and does not flush any of the buffer's contents to the disk. A return value of -1 indicates an error. If invalid parameters are passed, this function invokes the invalid parameter handler, as described in Parameter Validation. If execution is allowed to continue, the function returns -1 and **errno** is set to one of three values: **EBADF**, which means the file descriptor is invalid or the file is not opened for writing; **ENOSPC**, which means there is not enough space left on the device for the operation; or **EINVAL**, which means that *buffer* was a null pointer or that an odd *count* of bytes was passed to be written to a file in Unicode mode.

For more information about these and other return codes, see errno, _doserrno, _sys_errlist, and _sys_nerr.

If the file is opened in text mode, each linefeed character is replaced with a carriage return - linefeed pair in the output. The replacement does not affect the return value.

When the file is opened in Unicode translation mode—for example, if *fd* is opened by using **_open** or **_sopen** and a mode parameter that includes **_O_WTEXT**, **_O_U16TEXT**, or **_O_U8TEXT**, or if it is opened by using **fopen** and a mode parameter that includes **ccs=UNICODE**, **ccs=UTF-16LE**, or **ccs=UTF-8**, or if the mode is changed to a Unicode translation mode by using **_setmode**—*buffer* is interpreted as a pointer to an array of **wchar_t** that contains **UTF-16** data. An attempt to write an odd number of bytes in this mode causes a parameter validation error.

Remarks

The **_write** function writes *count* bytes from *buffer* into the file associated with *fd*. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer is increased by the

number of bytes actually written.

When writing to files opened in text mode, **_write** treats a CTRL+Z character as the logical end-of-file. When writing to a device, **_write** treats a CTRL+Z character in the buffer as an output terminator.

Requirements

ROUTINE	REQUIRED HEADER
_write	<io.h>

For additional compatibility information, see Compatibility.

Example

```
// crt__write.c
//
// This program opens a file for output and uses _write to write
// some bytes to the file.

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <share.h>

char buffer[] = "This is a test of '_write' function";

int main( void )
{
   int         fileHandle = 0;
   unsigned    bytesWritten = 0;

   if ( _sopen_s(&fileHandle, "write.o", _O_RDWR | _O_CREAT,
                 _SH_DENYNO, _S_IREAD | _S_IWRITE) )
      return -1;

   if (( bytesWritten = _write( fileHandle, buffer, sizeof( buffer ))) == -1 )
   {
      switch(errno)
      {
         case EBADF:
            perror("Bad file descriptor!");
            break;
         case ENOSPC:
            perror("No space left on device!");
            break;
         case EINVAL:
            perror("Invalid parameter: buffer was NULL!");
            break;
         default:
            // An unrelated error occured
            perror("Unexpected error!");
      }
   }
   else
   {
      printf_s( "Wrote %u bytes to file.\n", bytesWritten );
   }
   _close( fileHandle );
}
```

```
Wrote 36 bytes to file.
```

See also

Low-Level I/O
fwrite
_open, _wopen
_read
_setmode

wcsicoll

10/31/2018 • 2 minutes to read • Edit Online

This POSIX function is deprecated. Use the ISO C++ conformant _wcsicoll instead.

xor

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the ^ operator.

Syntax

```
#define xor ^
```

Remarks

The macro yields the operator ^.

Example

```
// iso646_xor.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a ^ b;
   cout << result << endl;

   result= a xor_eq b;
   cout << result << endl;
}
```

```
1
1
```

Requirements

Header: <iso646.h>

xor_eq

11/9/2018 • 2 minutes to read • Edit Online

An alternative to the ^= operator.

Syntax

```
#define xor_eq ^=
```

Remarks

The macro yields the operator ^=.

Example

```
// iso646_xor_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <iso646.h>

int main( )
{
   using namespace std;
   int a = 3, b = 2, result;

   result= a ^= b;
   cout << result << endl;

   a = 3;
   b = 2;

   result= a xor_eq b;
   cout << result << endl;
}
```

```
1
1
```

Requirements

Header: <iso646.h>

y0, y1, yn

10/31/2018 • 2 minutes to read • Edit Online

These POSIX functions are deprecated. Use the ISO C++ conformant Bessel Functions: _j0, _j1, _jn, _y0, _y1, _yn instead.