How to fit a million ✨ into a 📱

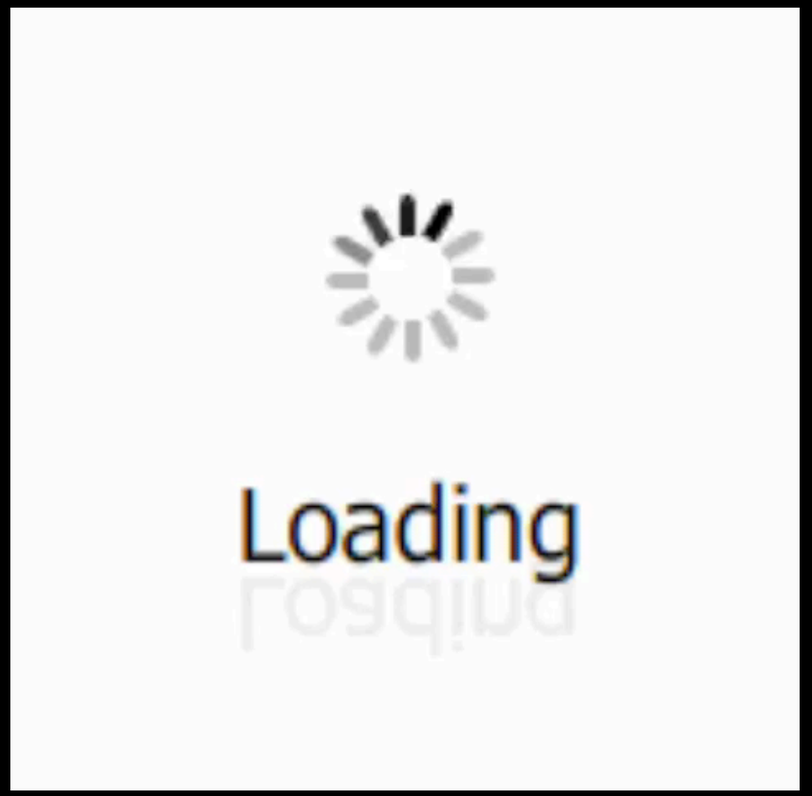
konrad@looc.io

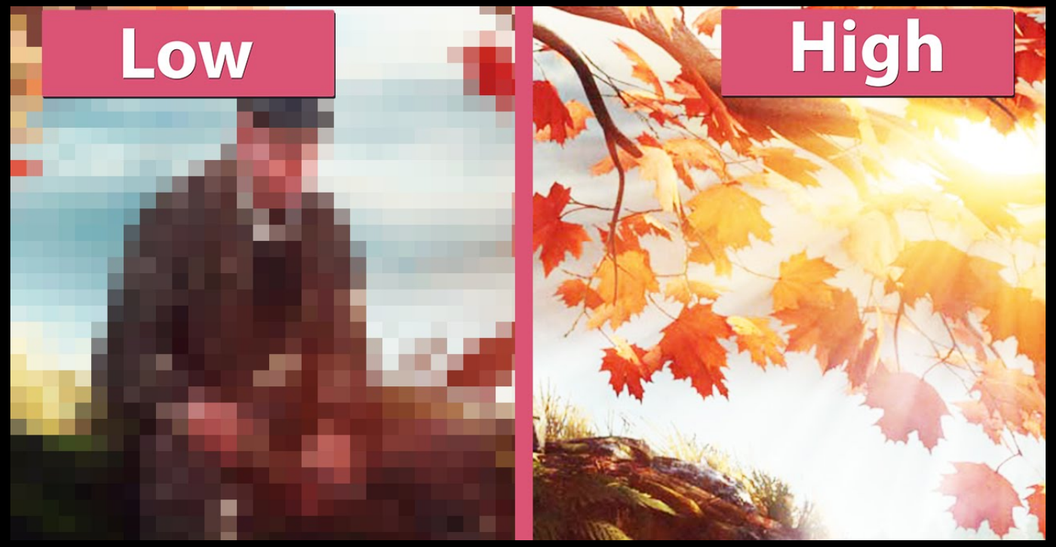
Content

- general points on optimization
- example problem
- some details on MemoryLayout of Structs in C/Swift
- Swift struct example optimization
- pretty results (hopefully)

Why Optimize Code?

- shorter loading times





- same features but better quality
  - graphics
  - sound
  - AI ...

Image: Vanishing of Ethan Carter © The Astronauts / Epic Games

Why Optimize Code?

* 
* every CPU cycle on mobile costs power (and $$$ when on server)
* sometimes 0fps -> good thing

Why Optimize Code?

- enable advanced features
- that weren't possible without optimization ☝️

⚡Premature Optimization⚡

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth

⚡Premature Optimization⚡

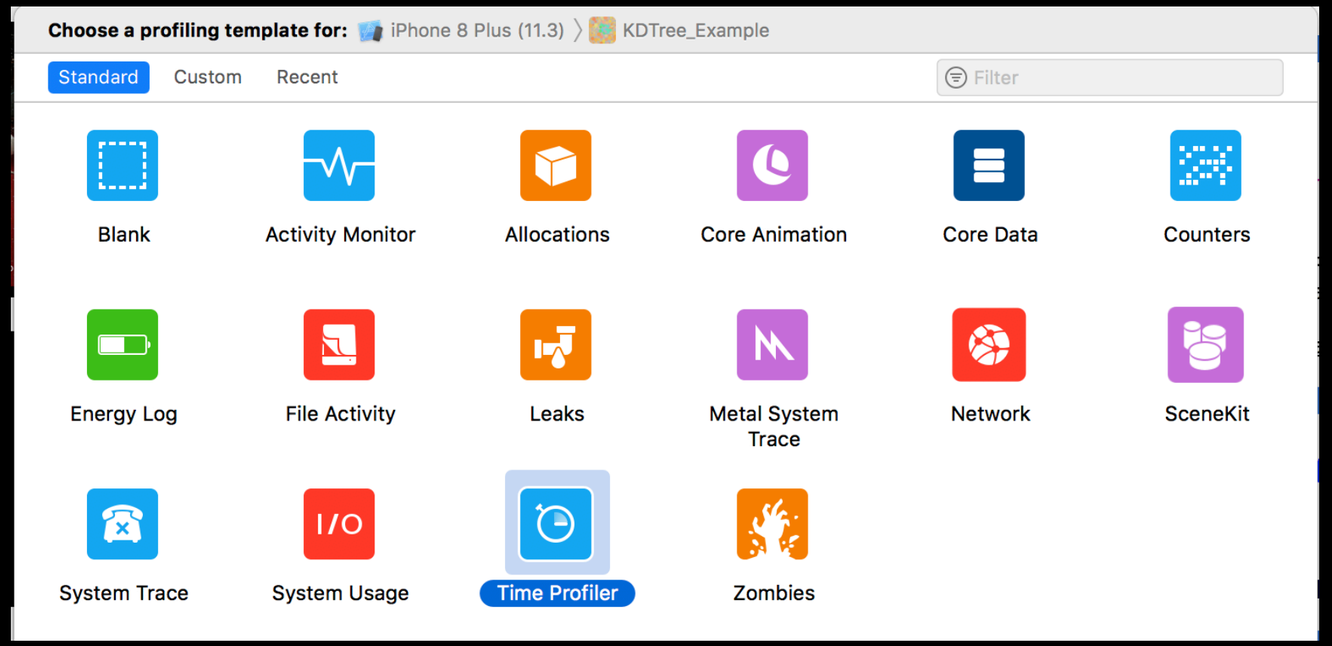
We want to write:

1. understandable, safe and testable code (DRY, KISS, etc.)
2. optimize as *needed*

- programming time is 💰
- faster code often more complex (💰)
- -> pick your battles

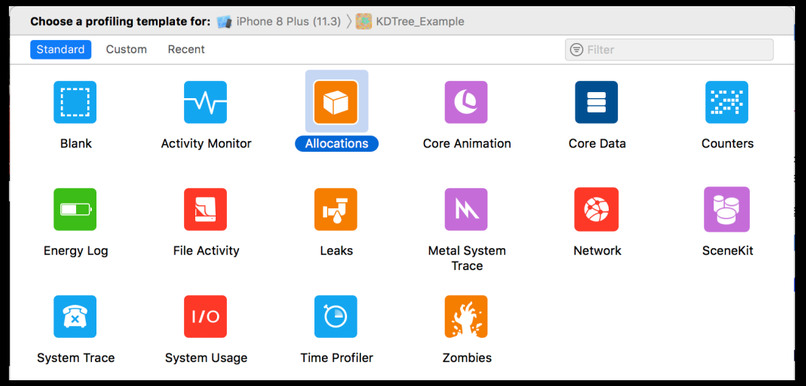
What to Optimize for?

- CPU cycles
- loading times, battery power, fps



What to Optimize for?

- memory
- how long our app stays in background mode before being killed by iOS
- fitting your project on small mobile devices (Watch!)



small memory size -> fast execution?

- Test example:

```
struct Element8 {
  let n: Int
}

struct Element16 {
  let n: Int
  let m: Int
}

struct Element32 { let k, l, m, n: Int }
struct Element48 { let k, l, m, n, o, p: Int }
// ...
struct Element512 { ... }
```

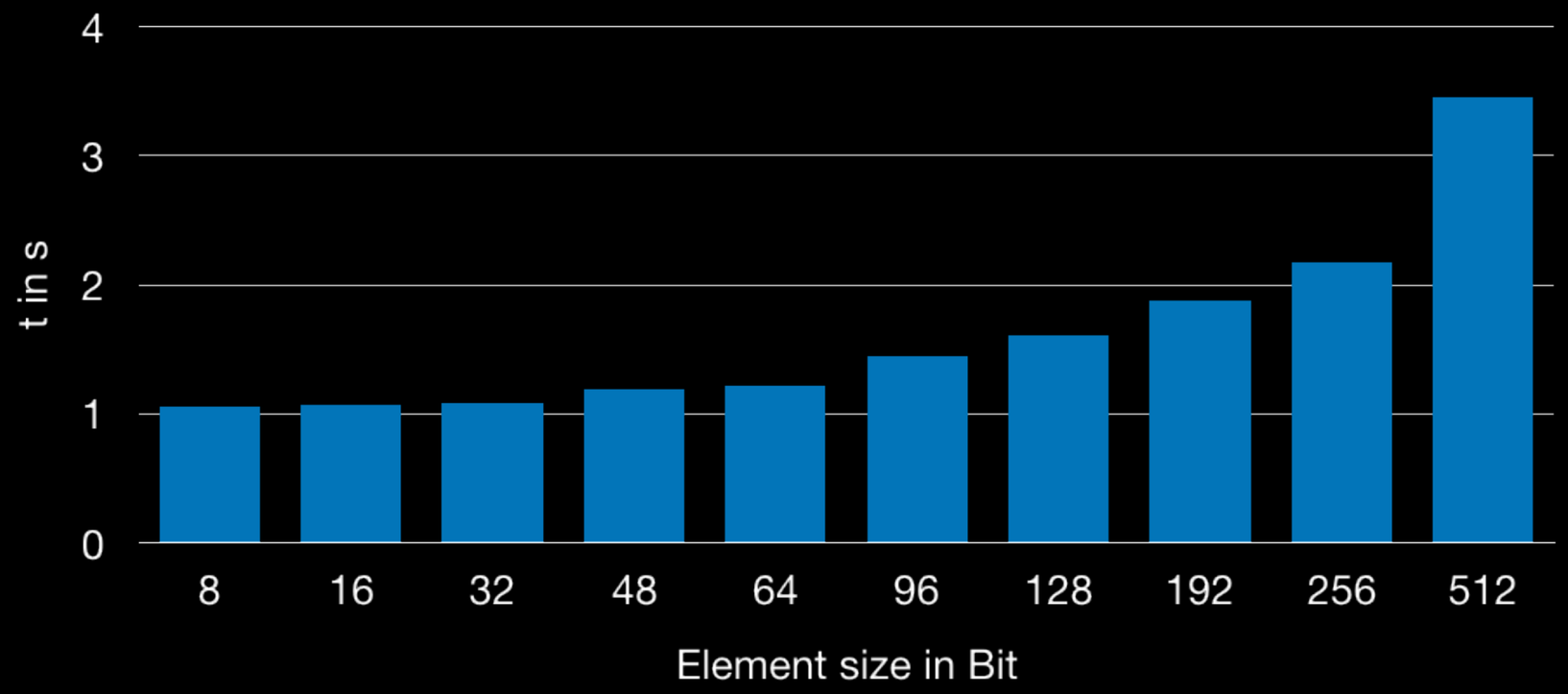
small memory size -> fast execution?

```
struct Element8 {
  let n: Int
}

let array: [Element] = // ...

measure {
  _ = array.reduce(0) { $0 + $1.n }
}
```

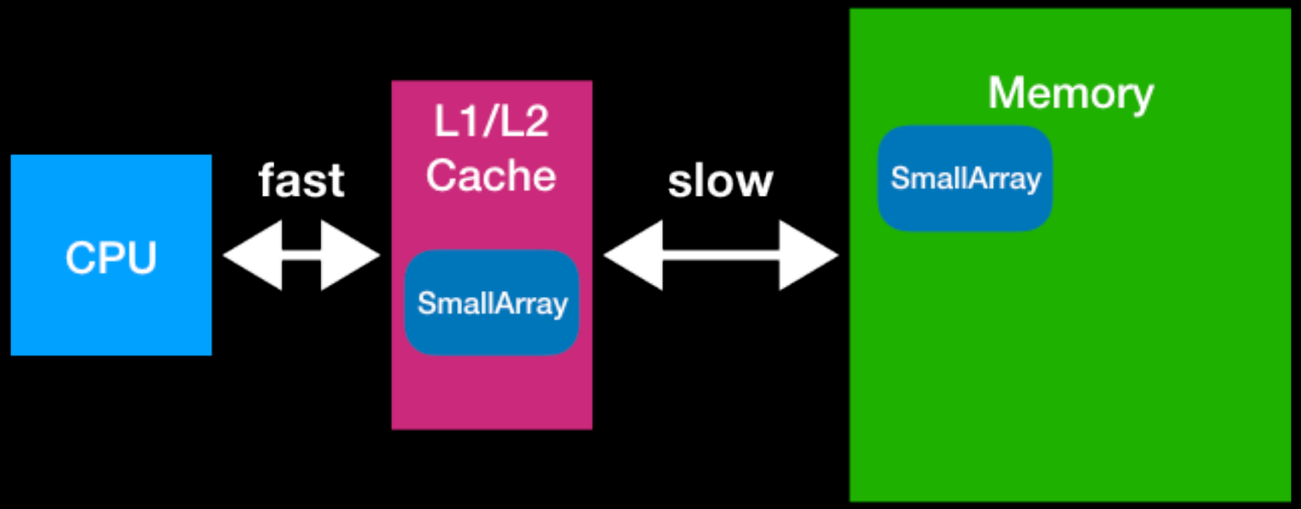
small memory size -> fast execution?



-
- every test has the same number of +-operations!

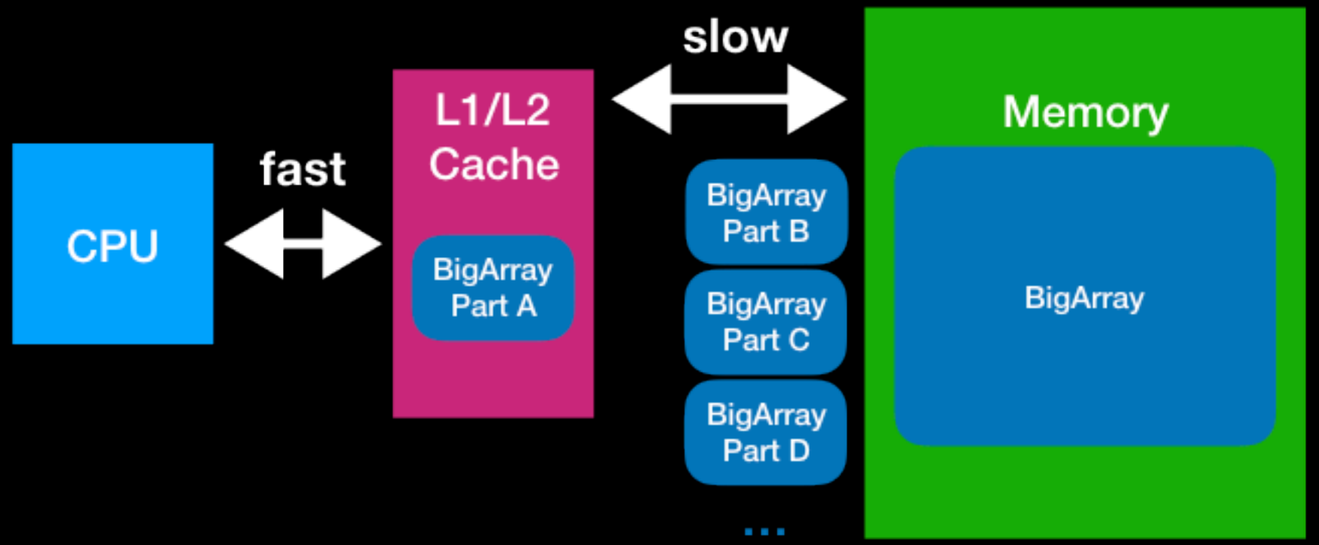
Explanation L1/L2 Caches

- small array has to be fetched from main memory in one piece



Explanation L1/L2 Caches

- large arrays have to be fetched in multiple steps



Sometimes CPU ⚡ Memory

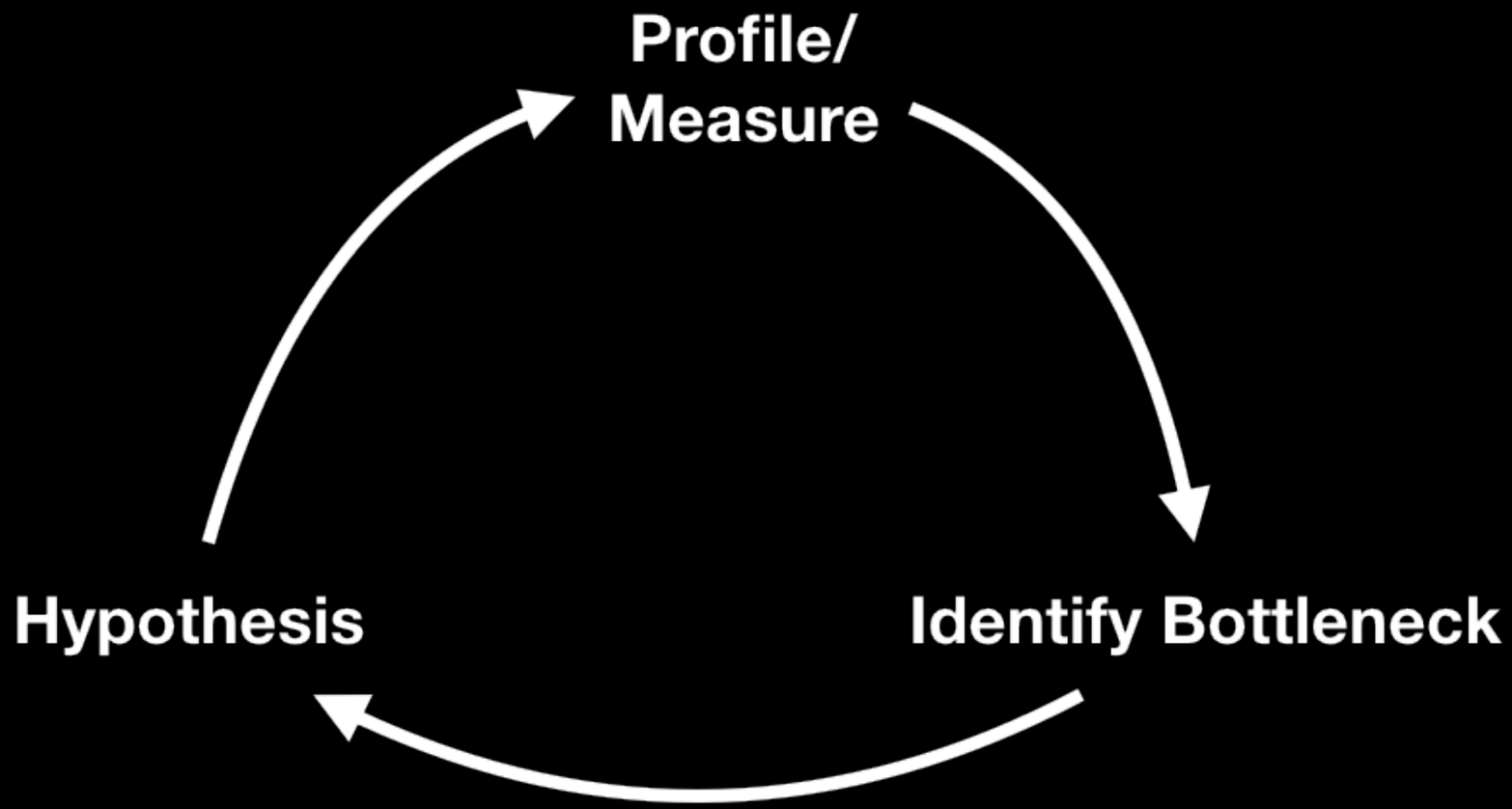
- e.g. all kinds of caches, simplest example:

```
lazy var lazilyCalculated: Object = {
  return makeHeavyObject()
}

// vs

var lazilyCalculated: Object {
  return makeHeavyObject()
}
```

Optimization Loop



Unit Tests!

```
func heavyCalculation(input: Input) -> Output {
  // lot's of code
  // ↻
}
```

- unit test `Input -> Output` for correctness
- optimize internal algorithm iteratively for performance

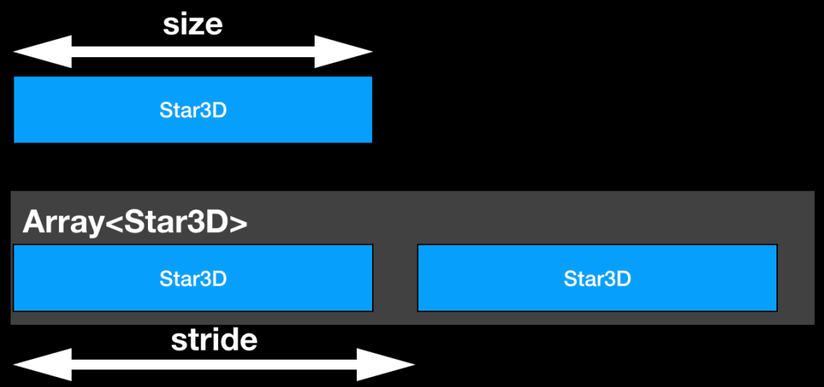
✨ Main Example: ✨

✨✨ Main Example: ✨✨

- large database of stars with many columns
- 8000 x ⭐ ~ 1.8 MB 🙂
- 120000 x ⭐ ~ 27 MB 😐
- 2.5mil x ⭐ ~ 560 MB 😣

Size vs Stride

- `MemoryLayout<Star3D>.size // 217`
- `MemoryLayout<Star3D>.stride // 224`



- $n * Stride(type) = Size(Array)$

Basic (Unoptimized?) Struct

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let hip_id: Int32?
    let hd_id: Int32?
    let hr_id: Int32?
    let gl_id: String?
    let bayer_flamstedt: String?
    let properName: String?
    let distance: Double
    let rv: Double?
    let mag: Double
    let absmag: Double
    let spectralType: String?
    let colorIndex: Float?
}

MemoryLayout<StarData>.stride // 208
```

How Big are Ints?

- `print(Int8.max) // 127`
- `print(Int16.max) // 32767`
- `print(Int32.max) // 2147483647`
- `print(Int.max) // 9223372036854775807`
- when designing the struct we already noticed we never need more than `Int32` to fit all stars

Basic (Unoptimized?) Struct

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let hip_id: Int32?          // 4
    let hd_id: Int32?           // 4
    let hr_id: Int32?           // 4
    let gl_id: String?
    let bayer_flamstedt: String?
    let properName: String?
    let distance: Double
    let rv: Double?
    let mag: Double
    let absmag: Double
    let spectralType: String?
    let colorIndex: Float?
}
```

Float

```
struct StarData {
    let right_ascension: Float    // 4
    let declination: Float        // 4
    let hip_id: Int32?            // 4
    let hd_id: Int32?             // 4
    let hr_id: Int32?             // 4
    let gl_id: String?
    let bayer_flamstedt: String?
    let properName: String?
    let distance: Double
    let rv: Double?
    let mag: Double
    let absmag: Double
    let spectralType: String?
    let colorIndex: Float?        // 4
}
```

Double

```
struct StarData {
    let right_ascension: Float    // 4
    let declination: Float        // 4
    let hip_id: Int32?            // 4
    let hd_id: Int32?             // 4
    let hr_id: Int32?             // 4
    let gl_id: String?
    let bayer_flamstedt: String?
    let properName: String?
    let distance: Double          // 8
    let rv: Double?               // 8
    let mag: Double               // 8
    let absmag: Double            // 8
    let spectralType: String?
    let colorIndex: Float?        // 4
}
```

String

```
struct StarData {
    let right_ascension: Float    // 4
    let declination: Float        // 4
    let hip_id: Int32?            // 4
    let hd_id: Int32?             // 4
    let hr_id: Int32?             // 4
    let gl_id: String?            // 24
    let bayer_flamstedt: String?  // 24
    let properName: String?       // 24
    let distance: Double          // 8
    let rv: Double?               // 8
    let mag: Double               // 8
    let absmag: Double            // 8
    let spectralType: String?     // 24
    let colorIndex: Float?        // 4
}
```

Quick Summing up of the Sizes

- Float, Int32 : 4 Byte
- Double: 8 Byte
- String: 24 Byte
- `4*24 + 4*8 + 6*4 = 152`
- 152 != 208
- 😳

Our Favorite Swift Type

```
enum Optional<Wrapped> {
    case none
    case some(Wrapped)
}
```

- Adds 1 Byte

```
MemoryLayout<Int>.size      // 8
MemoryLayout<Int?>.size     // 9
MemoryLayout<String>.size   // 24
MemoryLayout<String?>.size // 25
```

Removing Optionals

- let's use default value instead of nil (-1,"")
- stay safe by using `private` members and public getters

```
public func getHipId() -> Int? {
    return hip_id != -1 ? hip_id : nil
}
```

Struct Without Optionals

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let hip_id: Int32
    let hd_id: Int32
    let hr_id: Int32
    let gl_id: String
    let bayer_flamstedt: String
    let properName: String
    let distance: Double
    let rv: Double
    let mag: Double
    let absmag: Double
    let spectralType: String
    let colorIndex: Float?
}

MemoryLayout<StarData>.stride // 160
```

Struct Without Optionals

- wait, what?
- $208 - 160 \neq 9 * 1$
- we removed 9 Optionals and gained 48 Byte 🤔

Alignment (C Knowledge to the Rescue!)

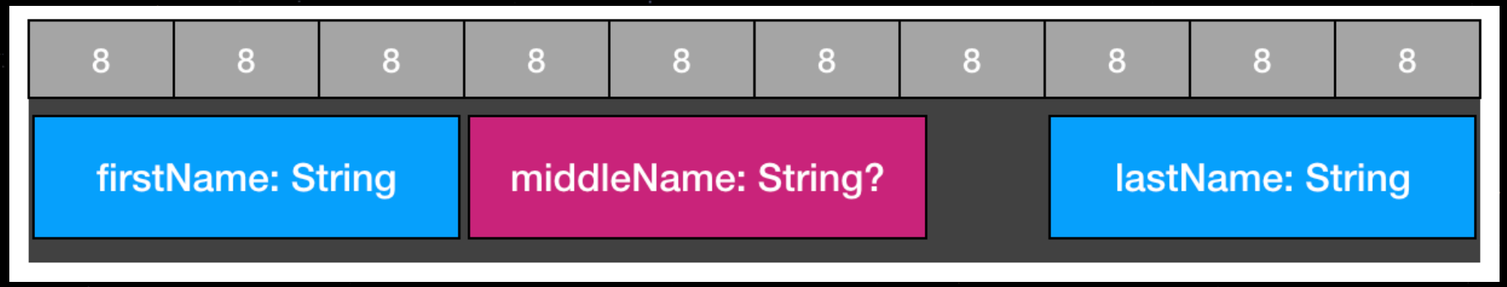
- modern CPUs lay out data types so memory access is fast
- each type has a *memory aligment*
  1. `char` can start anywhere
  2. `short` can start on even bytes
  3. `float` can start on 4/8/12/16/...
  4. ...

Alignment (Swift)

```
MemoryLayout<Int8>.alignment                 // 1
MemoryLayout<Int16>.alignment                // 2
MemoryLayout<Float>.alignment                // 4
MemoryLayout<Double>.alignment               // 8
MemoryLayout<Optional<Double>>.alignment // 8
MemoryLayout<String>.alignment               // 8
MemoryLayout<Optional<String>>.alignment // 8
```

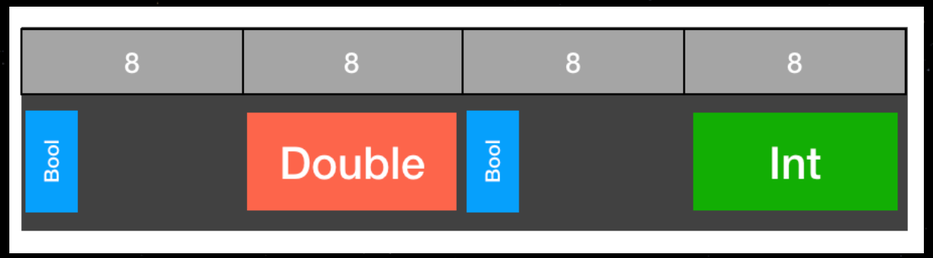
Padding

```
struct User {
  let firstName: String   // 24Byte starts at 0
  let middleName: String? // 25Byte starts at 24
                          // padding of 7 Byte
  let lastname: String    // 24Byte starts at 56.
}
MemoryLayout<User>.stride // 80
```



Bad Alignment Example

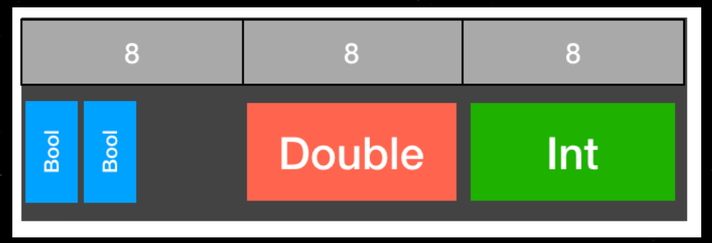
```
struct BadAligned {
    let isHidden: Bool
    let size: Double
    let isInteractable: Bool
    let age: Int
}
MemoryLayout<BadAligned>.stride // 32Byte
```



Better Alternative (saves 25%)

- generally, arrange from small -> large types

```
struct WellAligned {
    let isHidden: Bool
    let isInteractable: Bool
    let height: Double
    let age: Int
}
print(MemoryLayout<WellAligned>.stride) // 24
```



Aligned StarData

- 208 -> 152 (Optionals + Alignment)

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let hip_id: Int32
    let hd_id: Int32
    let hr_id: Int32
    let colorIndex: Float
    let distance: Double
    let rv: Double
    let mag: Double
    let absmag: Double
    let gl_id: String
    let bayer_flamstedt: String
    let properName: String
    let spectralType: String
}

MemoryLayout<StarData>.stride   // 152
```

Use Domain Knowledge

- largest piece are the strings
- turns out many strings are empty
- only 146 stars have proper names
- 3801 unique Gliese IDs
- 3064 Bayer Flamstedt Designations
- 4307 different spectral types (could be cleaned up)

Spare Strings into Separate Dictionaries

- during loading of database, index unique instances of strings
- create nice accessors to hide implementation detail

```
func getGlId() -> String? {
    return gl_id != -1 ? DB.glIds[Int(gl_id)] : nil
}
```

- But: dictionary-lookups means more work for cpu!
- This is one of those `Memory` ⚡ `CPU` examples we have to carefully profile

Spare Strings into Separate Dictionaries

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let hip_id: Int32
    let hd_id: Int32
    let hr_id: Int32
    let colorIndex: Float
    let distance: Double
    let rv: Double
    let mag: Double
    let absmag: Double
    let gl_id: Int16
    let bayer_flamstedt: Int16
    let properName: Int16
    let spectralType: Int16
}

MemoryLayout<StarData>.stride   // 64
```

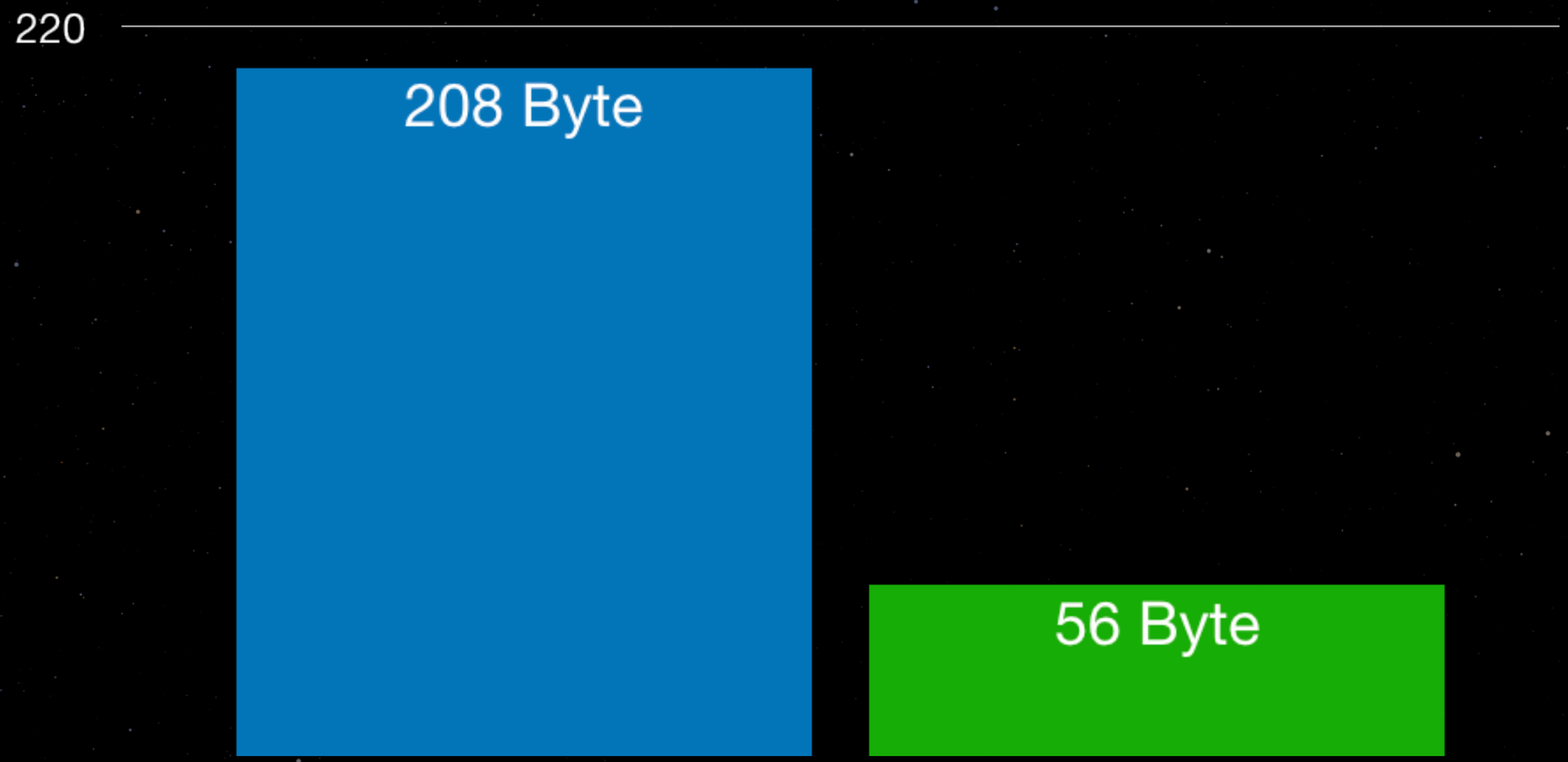
- are we done?

Alignment One More Time

```
struct StarData {
    let right_ascension: Float
    let declination: Float
    let spectralType: Int16
    let gl_id: Int16
    let bayer_flamstedt: Int16
    let properName: Int16
    let db_id: Int32
    let hip_id: Int32
    let hd_id: Int32
    let hr_id: Int32
    let rv: Float
    let mag: Float
    let absmag: Float
    let colorIndex: Float
    let distance: Double
}

MemoryLayout<StarData>.stride   // 56
```

Final Result



Reminder - Unit Test!

- while doing all the above optimizations:
- run unit tests after each change of code
- we can be sure we didn't break anything ⛑️

Premature optimization is the root of all evil...

Examples (when not to use what we just learned):

- set of 20 users ⚠️
- media library of 1000 movies ⚠️
- Server-Side Swift with > 10^6 entries ✅
- procedurally generated content in a game ✅
- points of interest in MapKit ✅

Extra: Swift Compiler

- we can also apply the same steps to swift project compilation time
- measure compilation of each function and sort result

```
xcodebuild -project App.xcproj -scheme App clean build
 OTHER_SWIFT_FLAGS="-Xfrontend -debug-time-function-bodies" |
 grep "[0-9][0-9]\.[0-9]*ms" | sort -nr > culprits.txt
```

```
430.23ms    ../Pods/SwiftyBeaver/Sources/AES256CBC.swift:356:18 instance method decrypt(block:)
267.29ms    ../LooC/ContrastTestViewModel.swift:99:22   instance method adjustBrightnessGradually()
262.38ms    ../LooC/HistoryViewModel.swift:215:18   instance method graphForNearVision(tests:firstT:lastT:)
250.10ms    ../LooC/HistoryViewModel.swift:276:10   instance method generateFakeTests()
224.70ms    ../LooC/HistoryViewModel.swift:221:41   (closure)
```

Server-Side Swift Example

- getting lots of data into a small machine works especially well on servers
- example: https://github.com/Bersaelor/StarsOnKitura / https://starsonkitura.eu-de.mybluemix.net

What did we learn today?

- pick your battles wisely!
- pack your struct's tightly ("play Tetris")
- know your domain to utilize its properties
- carefully balance CPU vs RAM
- profile first!

Links

- Premature Optimization http://wiki.c2.com/?PrematureOptimization
- The Lost Art of C Structure Packing http://www.catb.org/esr/structure-packing/
- Writing High-Performance Swift Code https://github.com/apple/swift/blob/master/docs/OptimizationTips.rst
- Swift Array Design https://github.com/apple/swift/blob/master/docs/Arrays.rst

Repos

- This talk & playgrounds: https://github.com/Bersaelor/Talks/tree/master/MillionStars-Moebius-Piter
- The 🌠-db: https://github.com/Bersaelor/SwiftyHYGDB
- KD-Tree structure: https://github.com/Bersaelor/KDTree

Thank you

- github.com/Bersaelor
- twitter.com/bersaelor