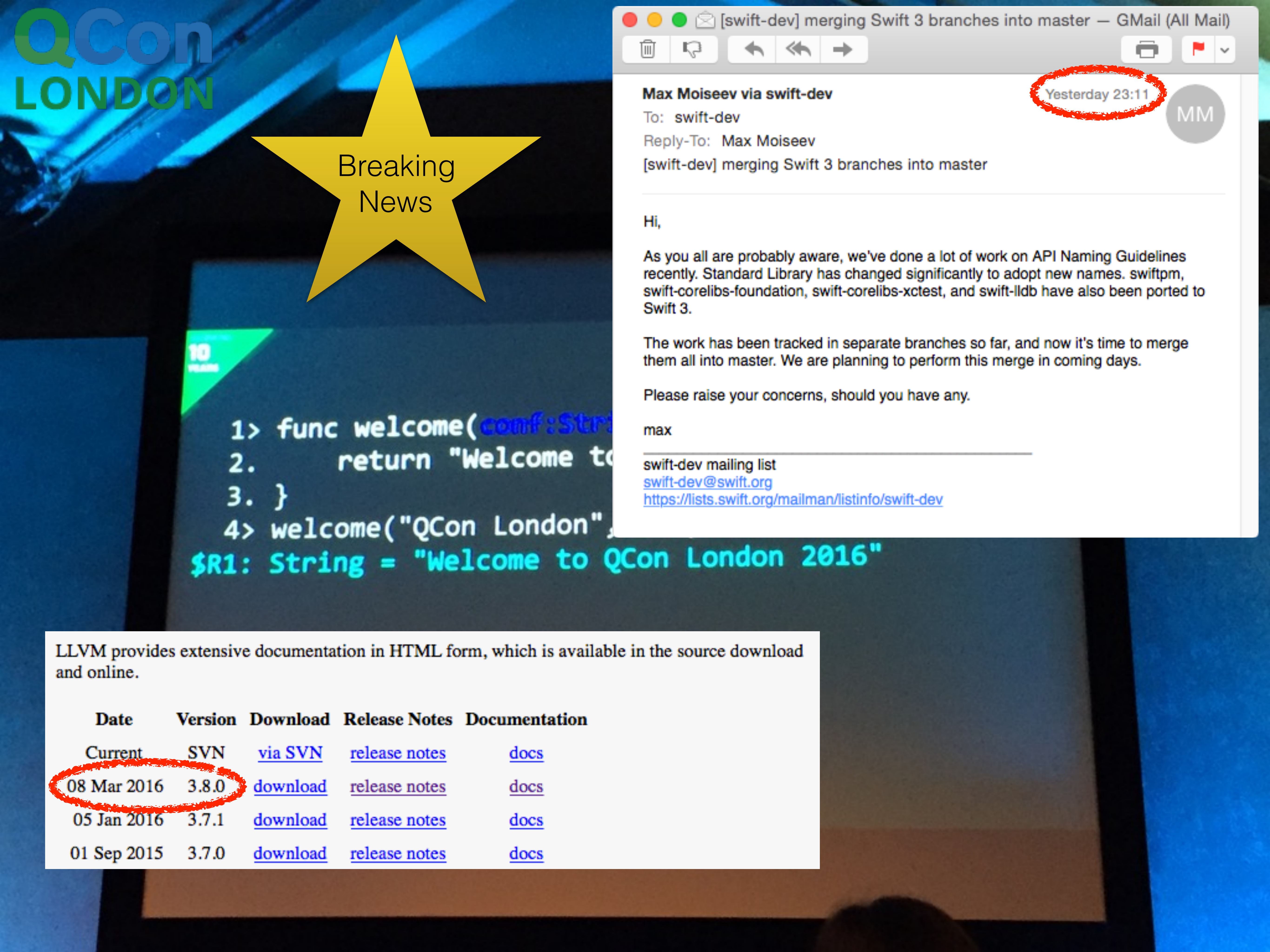
QCon LONDON

Swift 2 Under the Hood

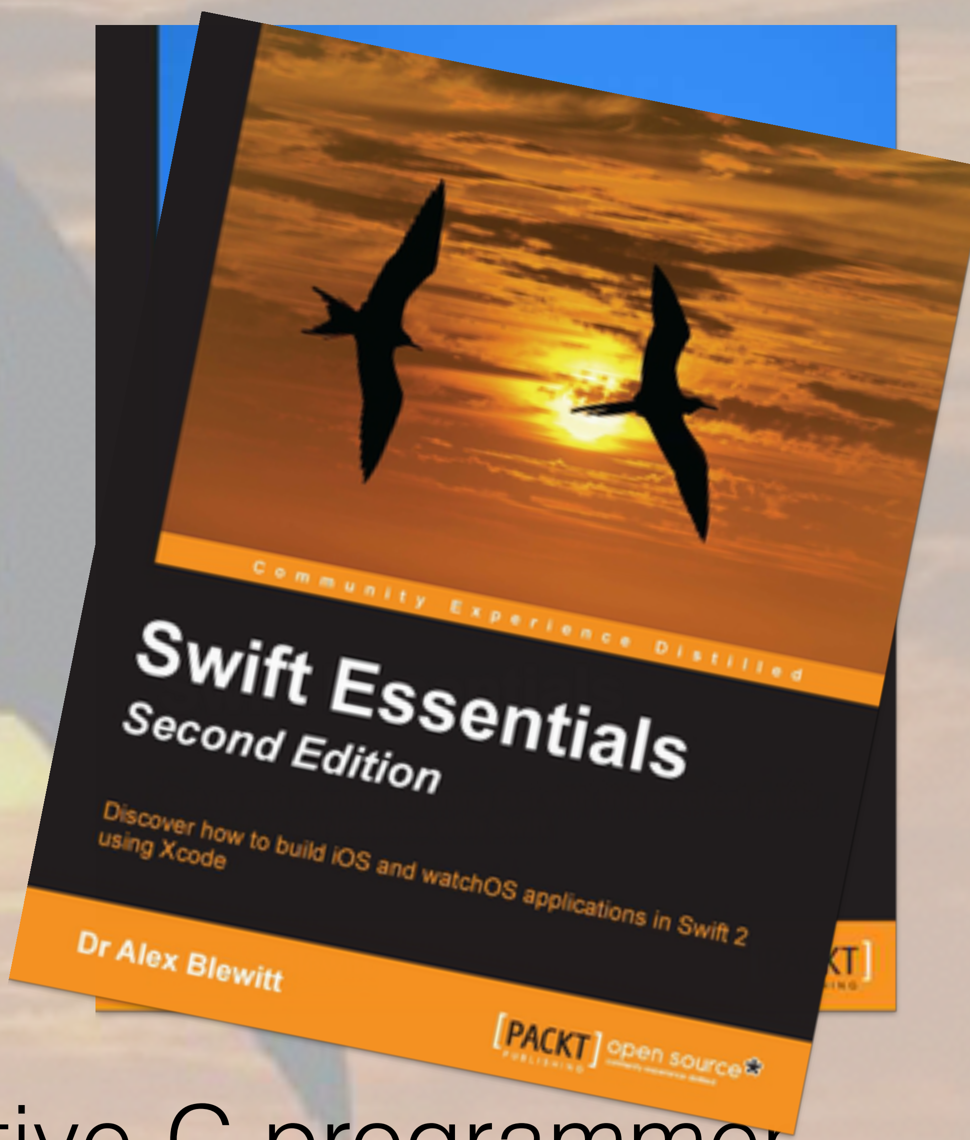

QCon
LONDON
Breaking News
[swift-dev] merging Swift 3 branches into master — GMail (All Mail)
Max Moiseev via swift-dev
Yesterday 23:11
MM
To: swift-dev
Reply-To: Max Moiseev
[swift-dev] merging Swift 3 branches into master
Hi,
As you all are probably aware, we've done a lot of work on API Naming Guidelines recently. Standard Library has changed significantly to adopt new names. swiftpm, swift-corelibs-foundation, swift-corelibs-xctest, and swift-lldb have also been ported to Swift 3.
The work has been tracked in separate branches so far, and now it's time to merge them all into master. We are planning to perform this merge in coming days.
Please raise your concerns, should you have any.
max
swift-dev mailing list
swift-dev@swift.org
https://lists.swift.org/mailman/listinfo/swift-dev
1> func welcome(
2. return "Welcome
3. }
4> welcome("QCon London",
$R1: String = "Welcome to QCon London 2016"
LLVM provides extensive documentation in HTML form, which is available in the source download and online.
Date Version Download Release Notes Documentation
Current SVN via SVN release notes docs
08 Mar 2016 3.8.0 download release notes docs
05 Jan 2016 3.7.1 download release notes docs
01 Sep 2015 3.7.0 download release notes docs

About This Talk

- Overview

> Based on Swift 2.x, the open source release in March 2016

  - Where did Swift come from?
  - What makes Swift fast?
  - Where is Swift going?
- Alex Blewitt @alblue
  - NeXT owner and veteran Objective-C programmer
  - Author of Swift Essentials https://swiftessentials.org

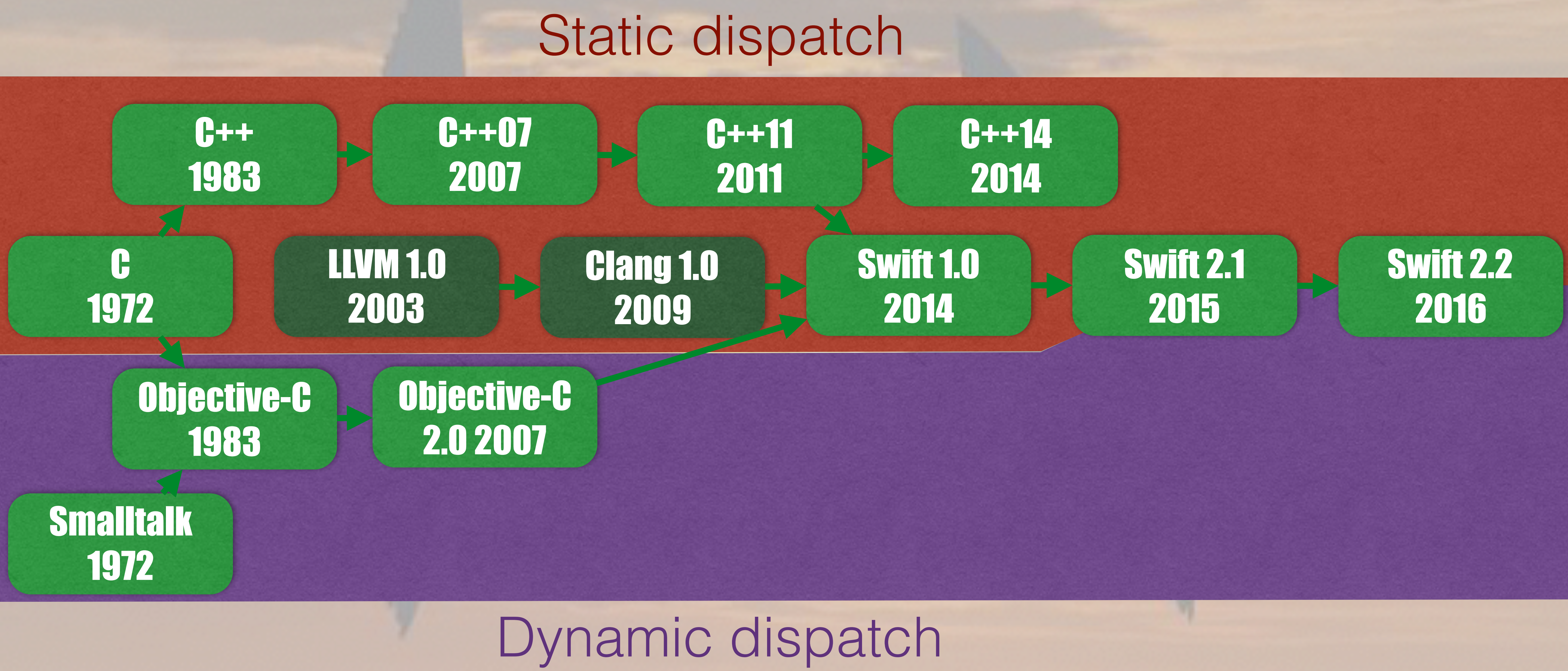


Where did Swift come from?

Pre-history

- Story starts in 1983 with Objective-C
  - Created as a Smalltalk like runtime on top of C
- NeXT licensed Objective-C in 1988
- NextStep released in 1989 (and NS prefix)
- Apple bought NeXT in 1996
- OSX Server in 1999
- OSX 10.0 Beta in 2000, released in 2001

Timeline

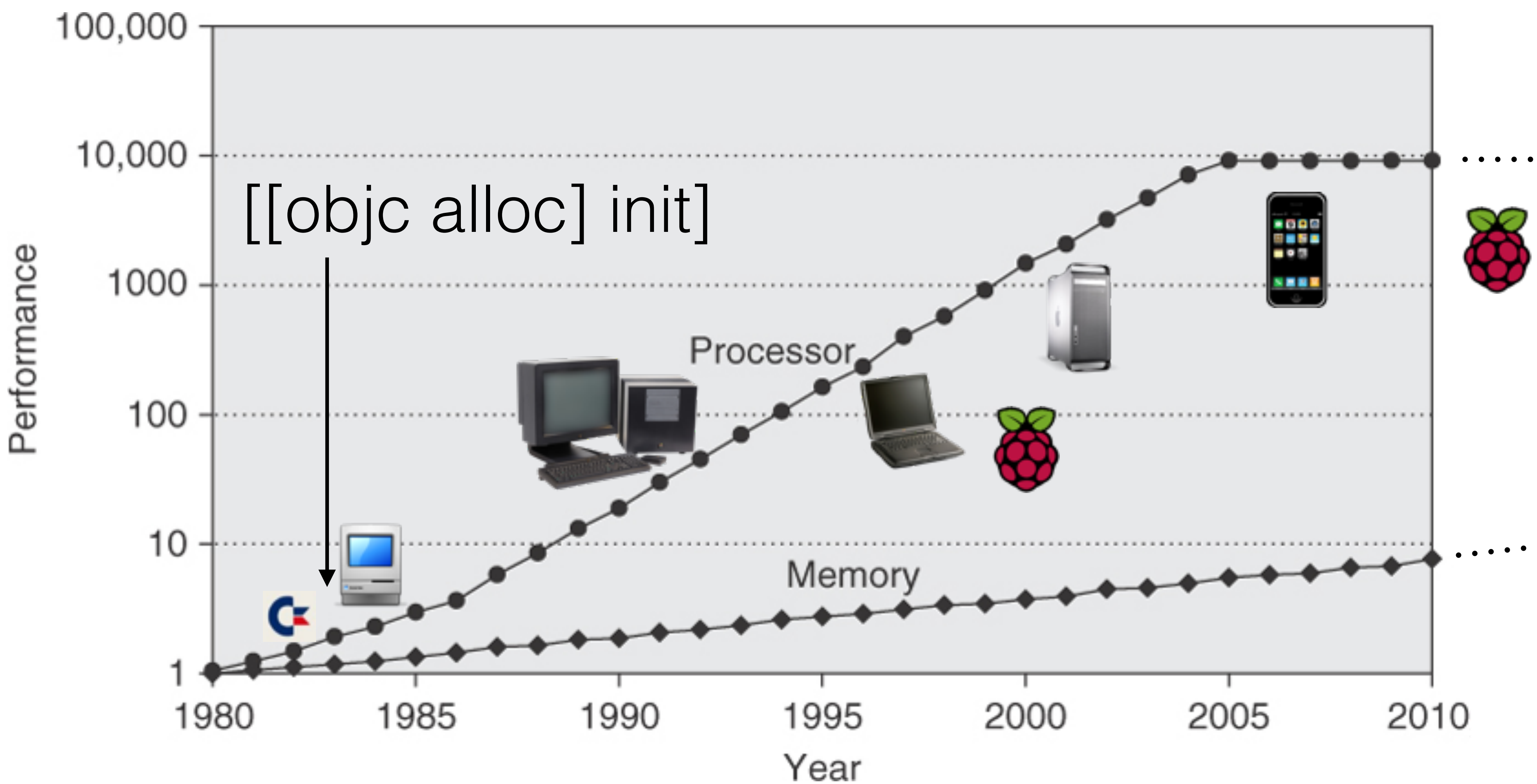

Static dispatch
C++ 1983
C++07 2007
C++11 2011
C++14 2014
C 1972
LLVM 1.0 2003
Clang 1.0 2009
Swift 1.0 2014
Swift 2.1 2015
Swift 2.2 2016
Objective-C 1983
Objective-C 2.0 2007
Smalltalk 1972
Dynamic dispatch

A lot has changed …

- CPU speed has risen for most of the prior decades
  - Plateaued about 3GHz for desktops
  - Mobile devices still rising; around 1-2GHz today
- More performance has come from more cores
  - Most mobiles have dual-core, some have more
  - Mobiles tend to be single-socket/single CPU
  - Memory has not increased as fast

QCon LONDON

CPU speed

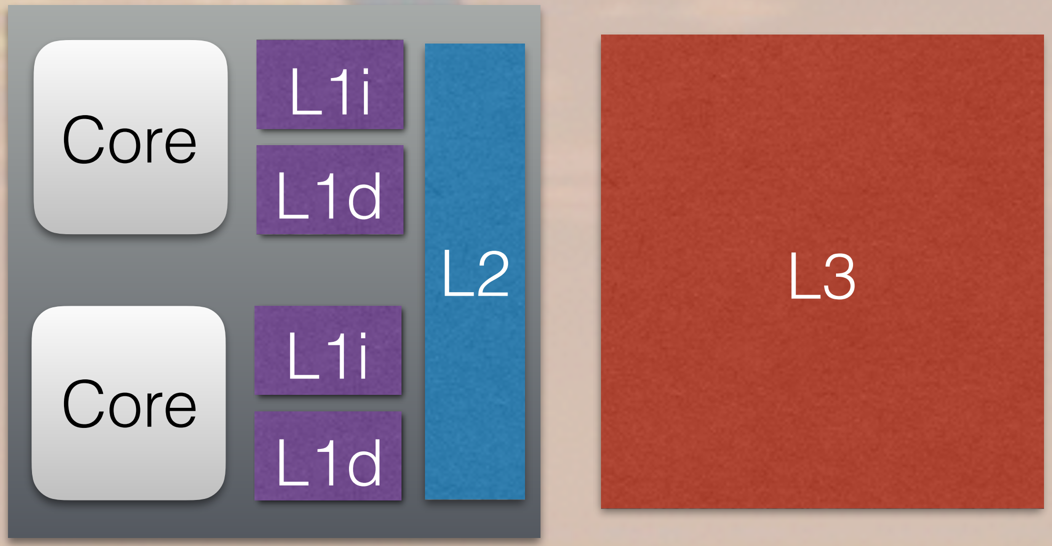


"Computer Architecture: A Quantitive Approach"
Copyright (c) 2011, Elsevier Inc
http://booksite.elsevier.com/9780123838728/

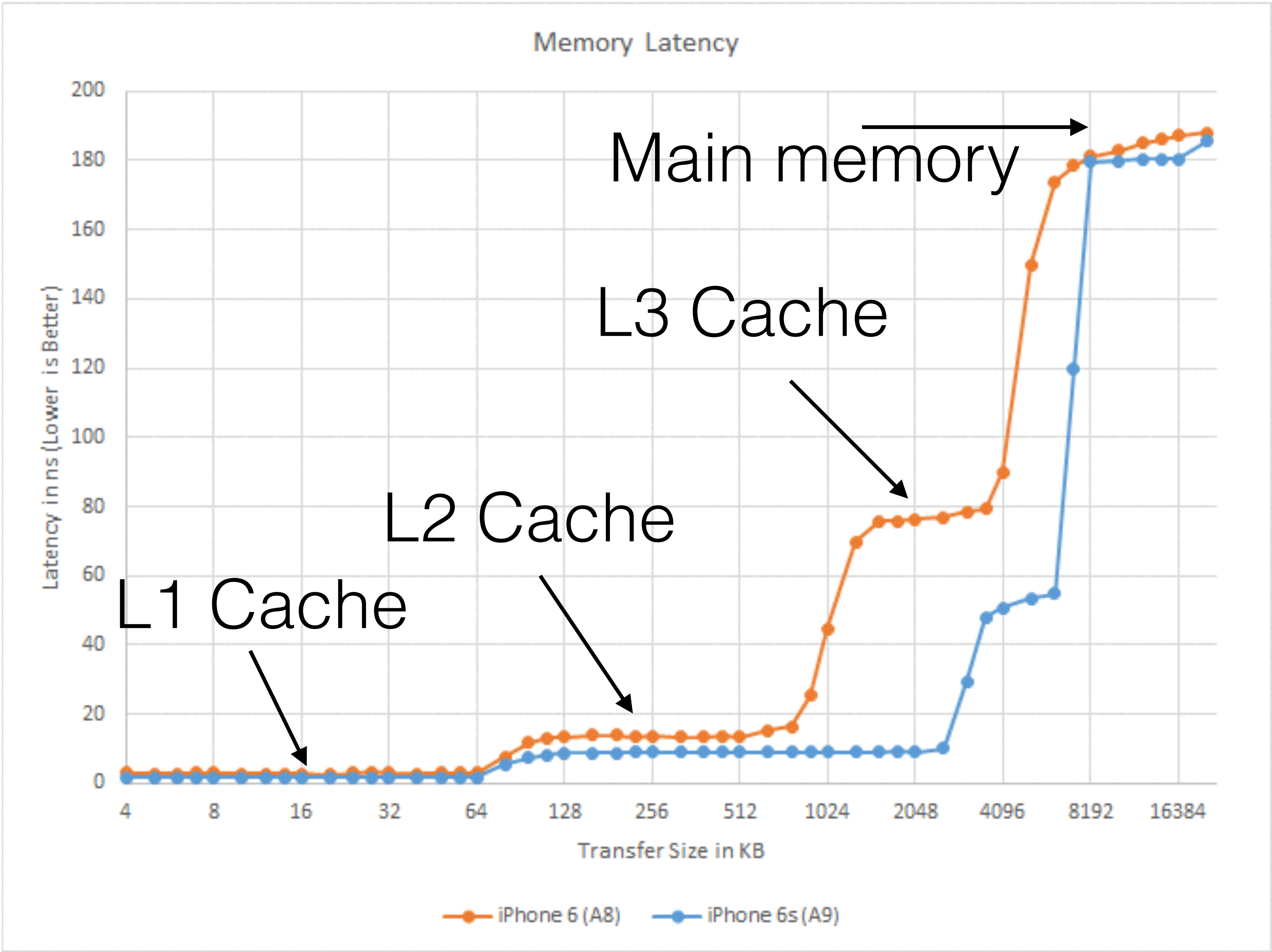
Memory latency

- Memory latency is a significant bottleneck
- CPU stores near-level caches for memory
  - L1 - per core 64k instruction / 64k data (~1ns)
  - L2 - 1-3Mb per CPU (~10ns)
  - L3 - 4-8Mb shared with GPU (~50-80ns)
- Main memory 1-2Gb (~180ns)

Numbers based on the iPhone 6 and iPhone 6s (A8 and A9)



Memory latency



AnandTech review of iPhone 6s
http://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/4

Why Swift?

Why Swift?

- Language features
  - Namespaces/Modules
  - Reference or Struct value types
  - Functional constructs
- Importantly
  - Interoperability with Objective-C
  - No undefined behaviour or nasal daemons

Modules

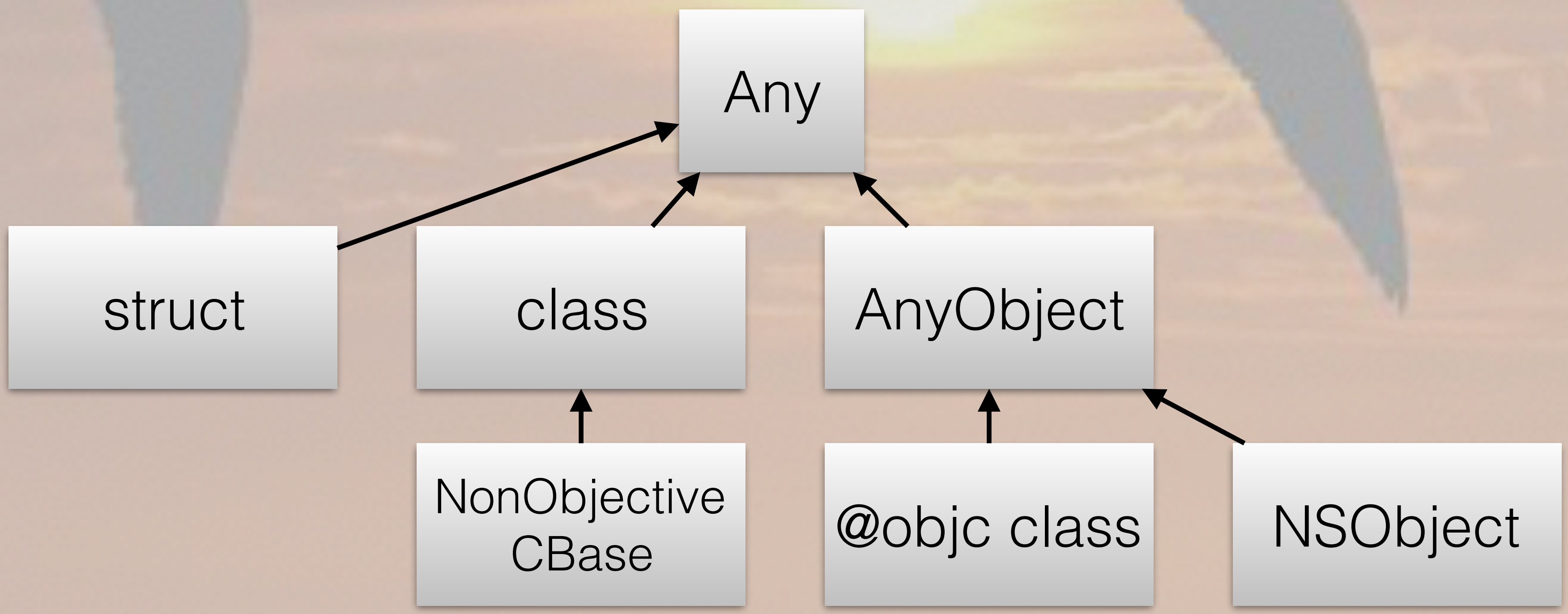
- Modules provide a namespace and function partition
- Objective-C
  - Foundation, UIKit, SpriteKit
- C wrappers
  - Dispatch, simd, Darwin
- Swift
  - Swift (automatically imported), Builtin

`Darwin` provides bindings with native C libraries e.g. `random()`

Builtin provides bindings with native types e.g. `Builtin.Int256`

Types

- Reference types: class (either Swift or Objective-C)
- Value types: struct
- Protocols: provides an interface for values/references
- Extensions: add methods/protocols to existing type



QCon LONDON

Numeric values

- Numeric values are represented as structs
  - Copied by value into arguments
  - Structs can inherit protocols and extensions

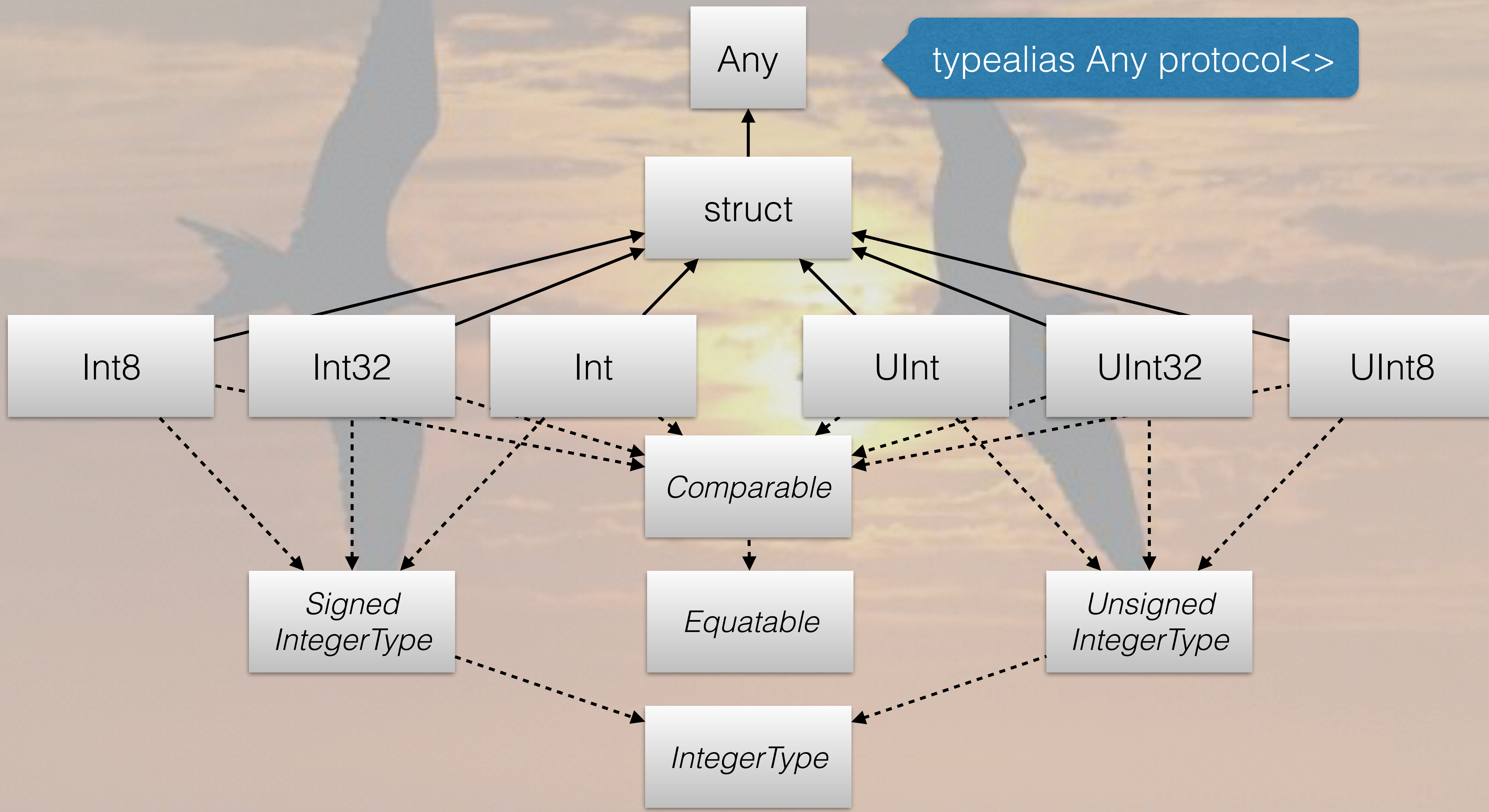
```
public struct Int : SignedIntegerType, Comparable {
  public var value: Builtin.Int64
  public static var max: Int { get }
  public static var min: Int { get }
}
public struct UInt: UnsignedIntegerType, Comparable {
  public var value: Builtin.Int64
  public static var max: Int { get }
  public static var min: Int { get }
}
```

sizeof(Int.self) == 8

sizeof(UInt.self) == 8

Protocols

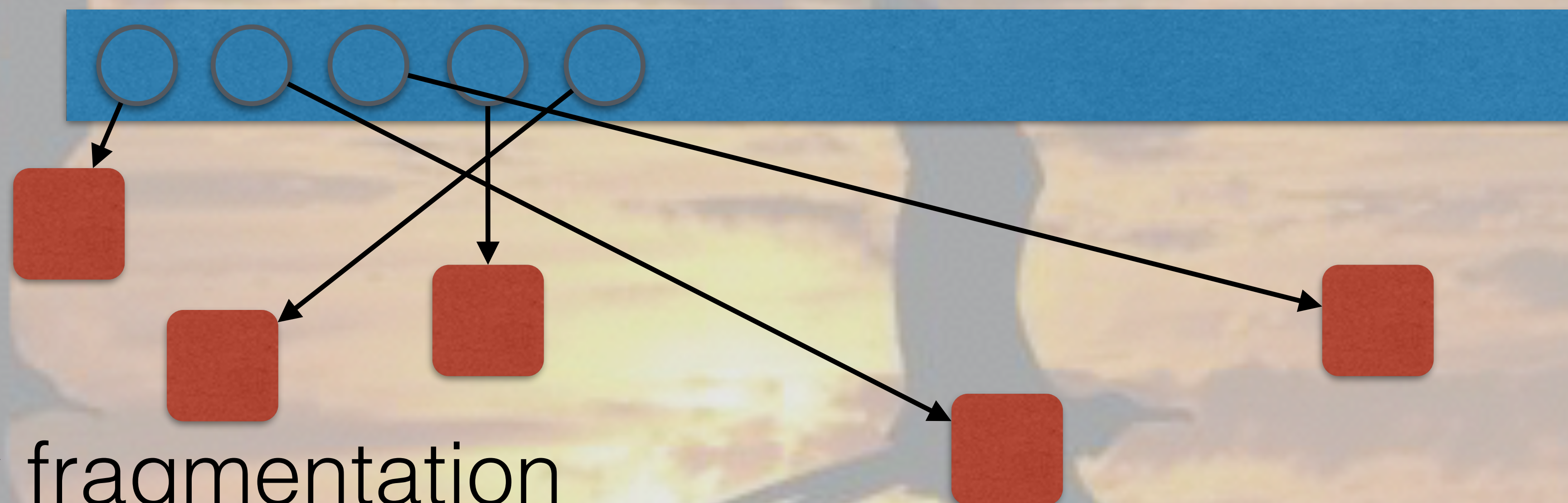
- Most methods are defined as protocols on structs



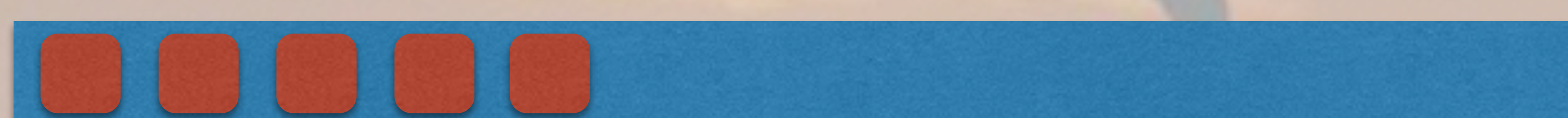
What makes Swift fast?

Memory optimisation

- Contiguous arrays of data vs objects
- `NSArray`



  - Diverse
  - Memory fragmentation
  - Limited memory load benefits for locality
- `Array<…>`



- Iteration is more performant over memory

Static and Dynamic?

- Static dispatch (used by C, C++, Swift)
  - Function calls are known precisely
  - Compiler generates `call`/`callq` to direct symbol
  - Fastest, and allows for optimisations
- Dynamic dispatch (used by Objective-C, Swift)
  - Messages are dispatched through `objc_msgSend`
  - Effectively `call(cache["methodName"])`

Swift can generate Objective-C classes and use runtime

Static Dispatch

```
a() -> b() -> c()
```

a ⟶ b ⟶ c

Optimises to **abc**

Dynamic Dispatch

```
[a:] -> [b:] -> [c:]
```

a ↘ ↗ b ↘ ↗ c

objc_msgSend objc_msgSend

Cannot be optimised

objc_msgSend

- **Every** Objective-C message calls `objc_msgSend`
  - Hand tuned assembly – fast, but still overhead

CPU, registers (`_cmd`, `self`), energy

Optimisations

- Most optimisations rely on inlining

  > Increases code size

  - Instead of `a() -> b()`, have `ab()` instead
  - Reduces function prologue/epilog (stack/reg spill)
  - Reduces branch miss and memory jumps
  - May unlock peephole optimisations
    - `func foo(i:Int) {if i<0 {return}…}`
    - `foo(-1)`

      > `foo(`*`negative`*`)` can be optimised away completely

Module Optimisation

- Whole Module Optimisation/Link Time Optimisation
  - Instead of writing out x86_64 .o files, writes LLVM
  - LLVM linker reads all files, optimises
  - Can see optimisations where single file cannot
- `final` methods and data structures can be inlined
  - Structs are always `final` (no subclassing)
  - `private` (same file) `internal` (same module)

QCon LONDON

Modules

- Swift performs separation through modules
  - `module.modulemap` standard Clang feature
  - http://clang.llvm.org/docs/Modules.html
- Provides a set of exports and runtime dependencies

```
module cryptoExample {
  requires cryptoCore
  header "cryptoExample.h"
  link "openssl"
  export *
}
module cryptoCore {
 …
}
```

Defines runtime dependencies

Adds -lopenssl or -framework foo

Can export subset of symbols

Building

- Swift **`build`** command used to build contents
  - Sources live in Sources, Source, src …
  - The **`PackageDescription`** module defines types
  - **`Package.swift`** used to define contents

```
 // example package
import PackageDescription
let package = Package(
  name: "CryptoPackage"
  dependencies: [
    .Package(url:"https://example.com/crypto.git",
      versions: Version(1,0,0)..<Version(2,0,0))
  ]
)
```

Platform code

QCon LONDON

- Conditional compilation using `#if` directives
  - Not implemented as a standalone pre-processor
  - Can also perform boolean operations ! || and &&
  - `os(OSX)`, `arch(i386)`, `DEBUG`, `ETC`

```
 // for OS specific code
#if os(Linux)
  import Glibc
#elseif os(OSX) || os(iOS)
  import Darwin
#endif
#if swift(>=1.2)
  …
#endif
```

Must be valid Swift syntax

QCon LONDON

Targets

- Swift project can generate multiple *targets*
  - `main.swift` results in command line tool
  - Otherwise module is named after parent directory
    - `Sources/crypto/secret.swift` -> secret.lib
    - `Sources/ad/other.swift` -> ad.lib
  - Can also describe `targets: [ Target(…) ]` in Package file

How does Swift work?

Swift and LLVM

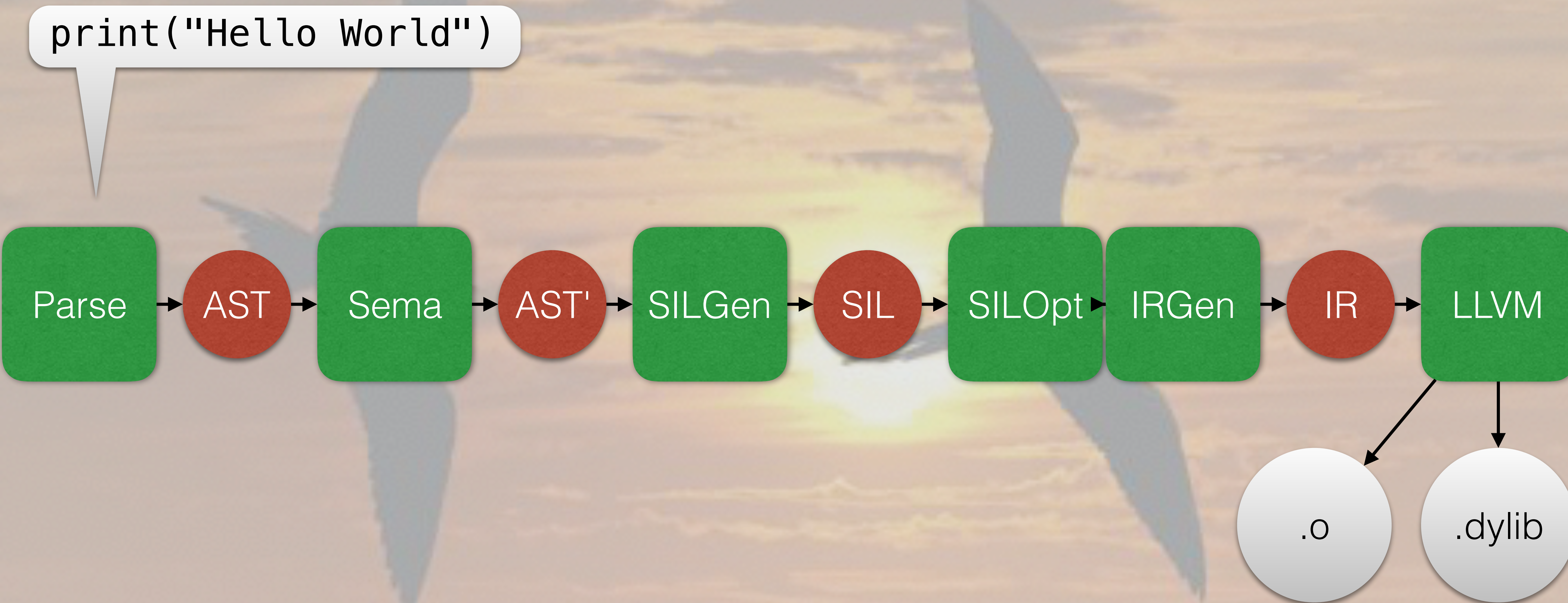
Bad name, wasn't really VMs

- Swift and clang are both built on LLVM
  - Originally stood for Low Level Virtual Machine
- Family of tools (compiler, debugger, linker etc.)
- Abstract assembly language
  - Intermediate Representation (IR), Bitcode (BC)
  - Infinite register RISC typed instruction set
  - Call and return convention agnostic

Swift compile pipeline

- AST - Abstract Syntax Tree representation
- Parsed AST - Types resolved
- SIL - Swift Intermediate Language, high-level IR
  - Platform agnostic (Builtin.Word abstracts size)
- IR - LLVM Intermediate Representation
  - Platform dependencies (e.g. word size)
- Output formats (assembly, bitcode, library output)

Swift compile pipeline



Example C based IR

- The ubiquitous Hello World program…

```
#include <stdio.h>

int main() {
  puts("Hello World")
}
```

```
clang helloWorld.c -emit-llvm -c -S -o -
```

```
@.str = private unnamed_addr constant [12 x i8] ⤢
 c"Hello World\00", align 1

define i32 @main() #0 {
  %1 = call i32 @puts(i8* getelementptr inbounds
   ([12 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

```
_main
  pushq %rbp
  movq  %rsp, %rbp
  leaq  L_.str(%rip), %rdi
  callq _puts
  xorl  %eax, %eax
  popq  %rbp
  retq
.section  __TEXT
L_.str:   ## was @.str
  .asciz  "Hello World"
```

main function

stack management

rdi = &L_.str

puts(rdi)

eax = 0

return(eax)

L_.str = "Hello World"

```
clang helloWorld.c -emit-assembly -S -o -
```

```
@.str = private unnamed_addr constant [12 x i8] ↩
 c"Hello World\00", align 1

define i32 @main() #0 {
  %1 = call i32 @puts(i8* getelementptr inbounds
   ([12 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Advantages of IR

- LLVM IR can still be understood when compiled
- Allows for more accurate transformations
  - Inlining across method/function calls
  - Elimination of unused code paths
  - Optimisation phases that are language agnostic

Example Swift based IR

- The ubiquitous Hello World program…

```
print("Hello World")
```

```
swiftc helloWorld.swift -emit-ir -o -
```

```
@0 = private unnamed_addr constant [12 x i8] ↩
 c"Hello World\00"

define i32 @main(i32, i8**) {
  …
  call void
   @_TFSs5printFTGSaP__9separatorSS10terminatorSS_T_
   %swift.bridge* %6, i8* %17, i64 %18, i64 %19,
   i8* %21, i64 %22, i64 %23)

  ret i32 0
}
```

QCon LONDON

Name Mangling

- Name Mangling is source → assembly identifiers
  - C name mangling: `main` → `_main`
  - C++ name mangling: `main` → `__Z4mainiPPc`
    - `__Z` = C++ name
    - `4` = 4 characters following for name (main)
    - `i` = int
    - `PPc` = pointer to pointer to char (i.e. `char**`)

QCon LONDON

Swift Name Mangling

- With the Swift symbol `_TFSs5printFTGSaP__9separatorSS10terminatorSS_T_`
  - `_T` = Swift symbol
  - `F` = function
  - `Ss` = "`Swift`" (module, as in `Swift.print`)
  - `5print` = "`print`" (function name)
  - `TGSaP___` = tuple containing generic array of Any (`[protocol<>]`)
  - `9separator` = "`separator`" (argument name, numeric prefix len)
  - `SS` = `Swift.String` (special case, as with other S* identifiers)
  - `T_` = empty tuple `()` (return type)

Swift Name Mangling

- With the Swift symbol `_TFSs5printFTGSaP__9separatorSS10terminatorSS_T_`

```
$ echo "_TFSs5printFTGSaP__9separatorSS10terminatorSS_T_" |
 xcrun swift-demangle

Swift.print ([protocol<>],
 separator : Swift.String,
 terminator : Swift.String) -> ()
```

- `5print` = "`print`" (function name)
- `TGSaP___` = tuple containing generic array protocol ([protocol<>])
- `9separator` = "`separator`" (argument name)
- `SS` = `Swift.String` (special case)
- `T_` = empty tuple `()` (return type)

Intermediate Language

- Swift IL Similar to LLVM IL, but with Swift specifics

```
print("Hello World")
```

```
swiftc helloWorld.swift -emit-sil -o -
```

```
sil_stage canonical

import Builtin
import Swift
import SwiftShims

// main
sil @main : $@convention(c) (Int32,
UnsafeMutablePointer<UnsafeMutablePointer<Int8>>) ->
Int32 {
  // function_ref Swift.print (Swift.Array<protocol<>>,
separator : Swift.String, terminator : Swift.String) ->
```

QCon LONDON

Swift vTables

- Method lookup in Swift is like C++ with vTable

```
class World { func hello() {…} }
```

```
swiftc helloWorld.swift -emit-sil -o -
```

```
 sil_stage canonical
import Builtin; import Swift; import SwiftShims
…
sil_vtable World {
 // main.World.hello (main.World)() -> ()
 #World.hello!1: _TFC4main5World5hellofS0_FT_T_

 // main.World.__deallocating_deinit
 #World.deinit!deallocator: _TFC4main5WorldD

 // main.World.init (main.World.Type)() -> main.World
 #World.init!initializer.1: _TFC4main5WorldcfMS0_FT_S0_
}
```

Default arguments

- The `print` function has default arguments
  - `separator` " " (between items)
  - `terminator` "\n" (at end)
- What does this do under the covers?

```
 // stdlib/public/core/Print.swift
public func print(
  items: Any...,
  separator: String = " ",
  terminator: String = "\n"
) {
```

An array of **Any** items

Printed between each item

Printed at end

Default arguments

- Each argument translated into function type
  - `Swift.print.defaultArgument1()`
  - `Swift.print.defaultArgument2()`
- In other words, `print("hello")` looks like:

```
 // stdlib/public/core/Print.swift
let sep = Swift.print.defaultArgument1()
let term = Swift.print.defaultArgument2()
apply(print,"Hello",sep,term)
/*
_TFSs5printFTGSaP__9separatorSS10terminatorSS_T_
_TIFSs5printFTGSaP__9separatorSS10terminatorSS_T_A0_
_TIFSs5printFTGSaP__9separatorSS10terminatorSS_T_A1_
*/
```

Errors

- Errors in Swift are denoted with `throws` and `try`
- The return result is wrapped up in a 2-tuple
    - (ordinary,error**)
- If error is non-null then error is raised

```
 // someFunc() throws -> Int32
try? someFunc()
// (ok,fail) = someFunc()
// return fail == null ? Optional(ok) : nil
try! someFunc()
// (ok,fail) = someFunc()
// return fail == null ? ok : fatalError()
do { try someFunc() } catch _ { }
// (ok,fail) = someFunc()
// if fail != null goto catch
```

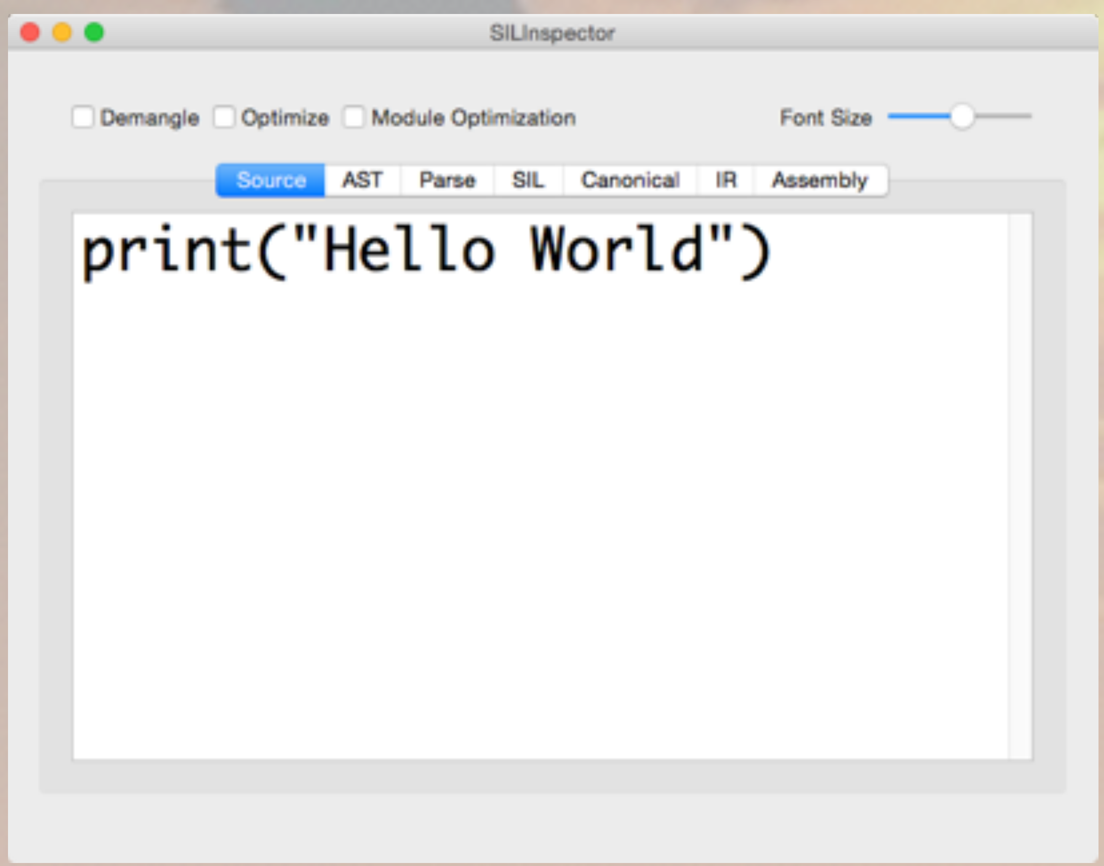
QCon LONDON

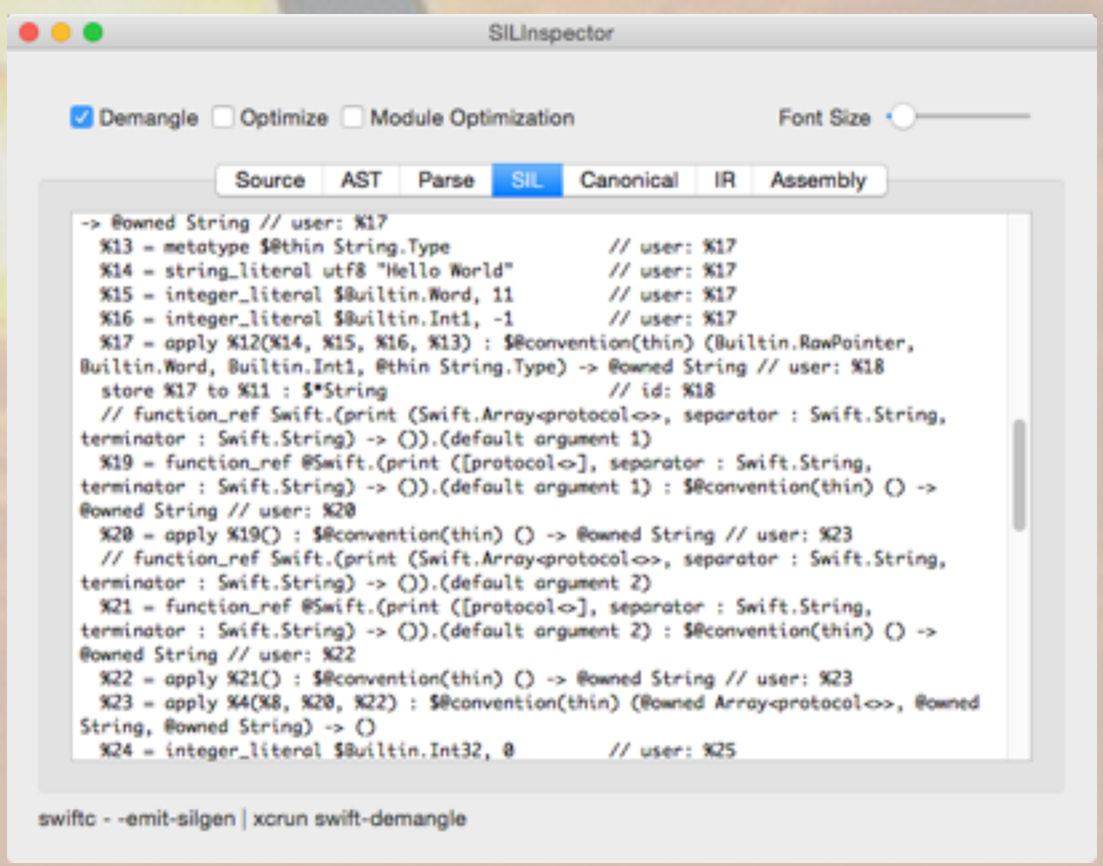
RefCounting

- Swift uses refcounting to free memory afterwards
- Copying variable increases ref count
- Memory freed when decreasing ref count to 0
- **@weak** required to avoid circular references
    - Difficult to have tooling to find this at the moment
    - Be aware of recursive cycles between objects
    - Break apart with **@weak** parent reference

SIL Inspector

- Allows Swift SIL to be inspected
- Available at GitHub
- https://github.com/alblue/SILInspector





QCon LONDON

SwiftObject and ObjC

- Swift objects can also be used in Objective-C
  - Swift instance in memory has an `isa` pointer
  - Objective-C can call Swift code with no changes
- Swift classes have `@objc` to use dynamic dispatch
  - Reduces optimisations
  - Automatically applied when using ObjC
    - Protocols, Superclasses

Swift advice

- Swift performance is changing – advice is out of date
- Default parameters result in additional function calls
- Embedded struct values can be performant in types
- Careful on hidden costs of passing structs to funcs
- Using `private` or `internal` allows for optimisations
- Avoid circular references counted class types


QCon
LONDON
Where is Swift going?
Open Source Swift 2 Under the Hood

Is Swift swift yet?

- Is Swift as fast as C?
  - Wrong question
- Is Swift as fast, or faster than Objective-C?
  - As fast or faster than Objective-C
  - Can be faster for data/struct processing
- More optimisation possibilities in future

Swift

- Being heavily developed – 3 releases in a year
- Provides a transitional mechanism from ObjC
  - Existing libraries/frameworks will continue to work
- Can drop down to native calls when necessary
- Used as replacement language in LLDB
- Future of iOS development?
- Future of server-side development?

Swift 3.0

- Next major release of Swift
- In preparation for late 2016 release
- Aims to provide (forward) binary compatibility
  - No more need to share source projects for modules
- Full generics
- API design guidelines and refactoring

Swift 3.0

- What it will not have:
  - Compatible with C++
  - Source compatible with Swift 2.x
    - Automated 'fix-ups' available for most features
  - Macros
  - Significantly different libraries in core

Changes to API

- API guidelines evolving to improve readability
  - `Type` suffix being removed (BooleanType)
  - `Generator` -> `Iterator`
- Mutators are imperative, non-mutators noun phrases
  - `sortInPlace()` -> `sort()`, `sort()` -> `sorted()`
  - `startsWith(_ prefix)` -> `starts(with: prefix)`
  - `minElement()` -> `min()` c.f. `max()`

Objective-C names

- Naming is evolving to avoid Objective-Cisms
  - `tableView(tableView: UITableView, numberOfRowsInsection section: Int) -> Int`
- Removing prefixes and repeated type information
- `String.fromCString()` -> `String(cString:)`
- `appendString(_: NSString)` -> `append(_: NSString)`

Swift 3.0

- Language is being designed in the open
  - Proposals vetted and voted in open
  - https://github.com/apple/swift-evolution
- Many community suggested improvements
  - Removal of ++ and -- operators, C for loops

QCon LONDON

Summary

- Swift has a long history coming from LLVM roots
- Prefers static dispatch but also supports objective-c
- Values can be laid out in memory efficiently
- In-lining leads to further optimisations
- Whole-module optimisation will only get better
- Modular compile pipeline allows for optimisations

QCon LONDON

Thanks

@alblue