



# Swift: Challenges and Opportunity for Language and Compiler Research

Chris Lattner, Apple Inc.  
IBM Programming Languages Day  
December 5, 2016

## Agenda

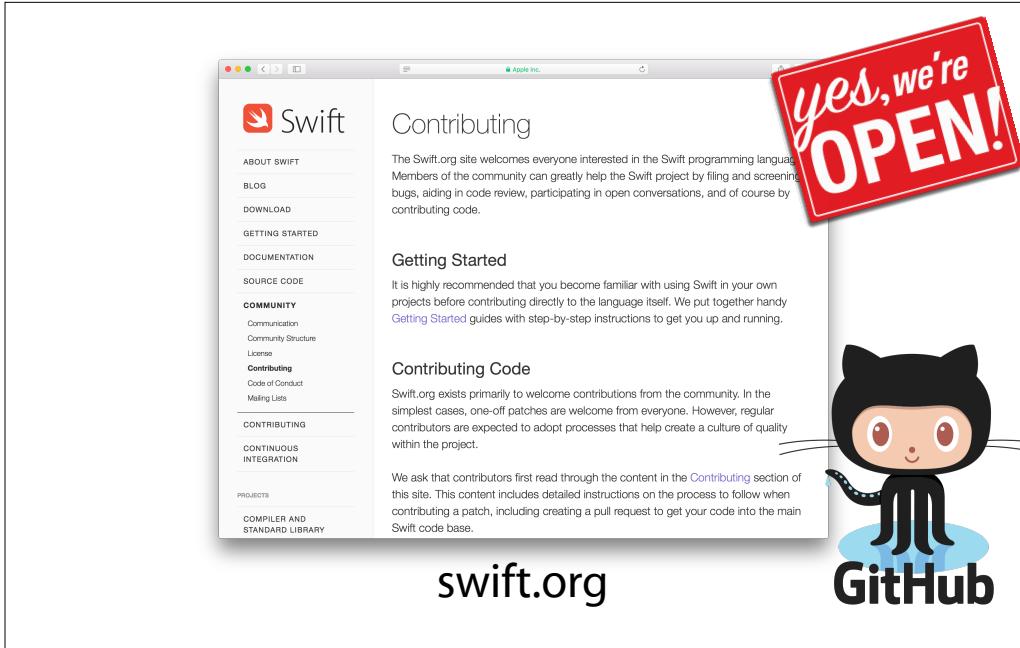
- Quick introduction to Swift
- Compilation model
- Automatic memory management
- Manual memory management
- Concurrency and reliability
- Language evolution

Swift has one common thread with Java: both languages invalidated many commonly held assumptions, which opens new avenues for investigation, and makes it perfect for academic and industrial research. Swift is just over 2 years old, so it is a great time to get in on the ground floor. There are a ton of great apps written in Swift, and thousands of developers actively using the language, with millions of lines of code in the world. And now, it is starting to branch out to new domains.

Focus of this talk is to explore interesting aspects of Swift's design (from the compiler/vm/runtime/languages perspective) and talk about a few major unexplored areas.

## What is Swift?

- Modern language with a familiar syntax
- Multiparadigm: functional, imperative, generic, OOP ...
- Aims to replace C family languages, among others
- Memory safe by default
- Implementation built on LLVM & Clang 
- Fully open source, portable to many platforms



- Fully open source, [swift.org](http://swift.org) opened its doors just over a year ago now.
- This includes open language evolution/design as well as implementation, and user community.
- The action takes place on github and in public mailing lists.

## Show me the code!

```
struct Point {  
    var x, y : Double  
  
}  
  
var p = Point(x: 12, y: 24)
```

Ok, I'll show you a quick example of some Swift code just so you have an example of the concrete syntax.

Swift has familiar syntax because it borrows good ideas from many other popular programming languages.

-> Explicitly typed, type inference, labeled arguments.

## Show me the code!

```
struct Point {  
    var x, y : Double  
  
    func scaled(by amount: Double) -> Point {  
        return Point(x: x*amount, y: y*amount)  
    }  
  
}  
  
var p = Point(x: 12, y: 24)  
let q = p.scaled(by: 0.5)
```

Functional programming features, methods can exist on all nominal types.

## Show me the code!

```
struct Point {
    var x, y : Double

    func scaled(by amount: Double) -> Point {
        return Point(x: x*amount, y: y*amount)
    }

    mutating func scale(by amount: Double) {
        x *= amount
        y *= amount
    }
}

var p = Point(x: 12, y: 24)
let q = p.scaled(by: 0.5)
p.scale(by: 1.5)
```

Designed for familiarity and pragmatism, not novelty or established dogma (e.g. allows explicit mutation). That said, the defaults encourage safety and predictability. For example, you have to add an extra keyword to enable mutability in structs.

Recurring theme: Swift enables multiple ways of doing things, but is opinionated about the “right default” which biases primarily towards safety/correctness, and secondarily towards performance.

## Kitura Web Framework Example

```
import Kitura

let router = Router()

router.get("/") { request, response, next in
    response.send("Hello, Server Side Swift")
    next()
}

Kitura.addHTTPServer(onPort: 8090, with: router)
Kitura.run()
```



Here is a more realistic example using IBM's Kitura web framework. You can see that Swift feels a lot like a scripting language, while provides performance of a low-level programming language. This example shows a trailing closure, similar to what you'd see in Ruby.

## **“The Swift Programming Language” Book**



<https://swift.org/documentation/>

This isn't a talk aimed to give an exhaustive intro to Swift, but there is free online documentation and even a complete ePUB book available on the Apple iBooks Store or from swift.org.

## Designed for Generality

- Applications
- Servers
- Systems
- Scripting
- Education



Many languages start out in a niche, and grow from there. In some cases, this leads to challenges when the community inevitably grows and begins applying it to new domains (e.g. scalability problems when writing large scale Javascript apps).

App programming: every language needs a killer app to get early traction. iOS App Programming was clearly Swift's first killer app that launched it to have a very active and amazing developer community.

That said, our goal for Swift has always been for it to “take over the world,” by which I mean that we hope that it will be as popular as Java has grown to be over the last 20 years. To enable this, we’re focusing on Swift’s success in the long term, and carefully building out core pieces of the language and eco system, so that 20 years from now, Swift is still a great language and system.

While we’re going to spend most of the talk discussing **complex** parts of the language, I want to emphasize that a huge amount of focus in Swift is to enable simplicity.

## Education

```
print("hello Swift")
```

```
$ swift hello.swift  
hello Swift  
$
```

Swift was intentionally designed to allow **progressive disclosure of complexity**. This is a full Swift program.

- See that it has no semicolons, and no \n. Truly spent a ton of effort to make the most minimal Swift programs distraction free.
- Compare this to Hello World in Java, one of the most successful languages for teaching in colleges, which requires a ton of boiler plate, and intense number of concepts.
- Just like a scripting language, you can run a swift Script like this.

## Education

```
#!/usr/bin/swift  
print("hello Swift")
```

```
$ swift hello.swift  
hello Swift  
$
```

```
$ chmod u+x ./hello.swift  
$ ./hello.swift  
hello Swift  
$
```

You can even add a `#!` line to your file, and directly run it, just like you'd do a shell script.

## Education

```
#!/usr/bin/swift  
print("hello Swift")
```

```
$ swift hello.swift  
hello Swift  
$
```

```
$ chmod u+x ./hello.swift  
$ ./hello.swift  
hello Swift  
$
```

```
$ swift  
Welcome to Apple Swift version 3.0.1. Type :help for assistance.  
1> 1 + 2  
$R0: Int = 3  
2>
```

And finally, Swift even includes a built in REPL, so if you'd like to hack out concepts on the command line, you can dive right in.

Enabling Swift to take off in the Education space isn't a small effort either...

## Swift Playgrounds for iPad



Apple recently shipped an entire App for the iPad, called “Swift Playgrounds” and a set of curriculum named “Learn to Code”. It builds on this minimality of code enabled by the Swift language design to teach programming to folks who have never programmed before.

We hope that teaching coding to people who otherwise wouldn’t have encountered it can help capture the curiosity and excitement of the next generation, and we’re continuing to invest in this area.

# Simple Statements

```
turnLeft()  
moveForward()  
turnLeft()  
moveForward()  
collectGem()
```

Playgrounds uses real Swift code, the whole time, from the first puzzle you encounter.  
Progressive disclosure of complexity is key. You can start with simple function calls to teach basic computational thinking.

## Logic and Control Flow

```
for i in 1 ... 6 {  
    moveForward()  
    if isOnGem() {  
        collectGem()  
    }  
}
```

“Learn to Code” then introduce new concepts one at a time, and uses puzzles to challenge the learner to use the new concepts to solve problems. Here we show basic looping and conditionals.

## Procedural Abstraction

```
func turnRight() {  
    turnLeft()  
    turnLeft()  
    turnLeft()  
}  
  
moveForward()  
turnRight()  
moveForward()  
collectGem()
```

Here is an example used to introduce functions, showing how to build your own abstractions. Swift's huge advantage here is that it gets boilerplate and other syntax noise out of the way. This is key for people learning, but it happens to also be highly valued by experienced Swift developers!

Swift Playgrounds is a free download on iPads and is frequently updated, please check it out if you're interested in learning more.

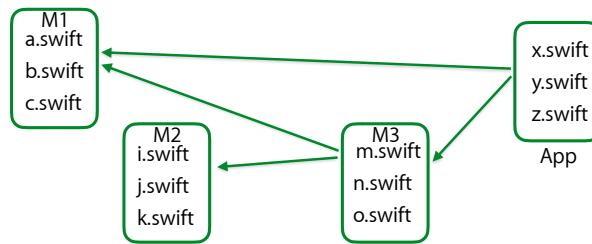
Despite its focus on simplicity, Swift has surprising depth, and the implementation is surprisingly complex in some places. I'd like to dive into some of the hard parts now, and talk about some open projects.

# Swift Compilation Model

Lets start by talking about how the Swift compiler works.

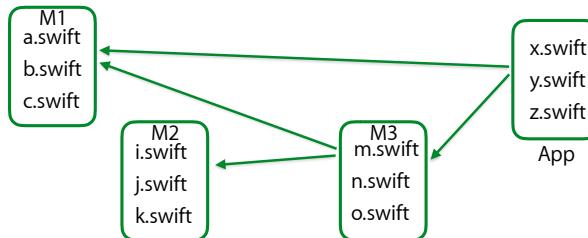
## Swift Program Structure

- Swift programs are made out of a DAG of modules
- Modules:
  - contain many files, which cross reference each other
  - are separately compiled in dependence order
  - are potentially very large



## Challenges to the Compilation Model

- Fast compile times:
  - Both clean and incremental builds
- Fast generated code:
  - Generics specialization, inlining, devirtualization, ...
- Resilience:
  - Allow modules to be separately compiled and separately evolving



Resilience: We don't want things like the C++ “fragile base class” problem, or to require all Swift code to be compiled from scratch any time a low-level module’s implementation details change.

For example, an app that provides an API for plugins to use shouldn’t break 3rd party binary plugins when the app changes.

## Static Compilation

- Obvious benefits:
  - Fast application startup time
  - Simple and deterministic runtime
  - Application has low resident set size
- Imposes constraints on language design:
  - No profile feedback / dynamic recompilation
  - Pervasive polymorphism ⇒ bad performance
- Non-obvious benefits:
  - Predictable performance model for programmer
  - Doesn't preclude use of a JIT compiler

Static compilation (as opposed to JIT compilation) is the default assumption of the Swift language.

Not all JIT compiled languages can be effectively statically compiled (e.g. due to things like class loading in Java, “eval” in scripting languages, etc), but all statically compiled languages can be JIT compiled.

This is because a JIT has strictly more information available to it than a static compiler. For more of a deep dive in this area, see this email: <https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20151207/001948.html>

# Swift Compiler Design



The Swift compiler follows common patterns in compiler design, starting with a parser and semantic analysis to build a typed AST.

# Swift Compiler Design

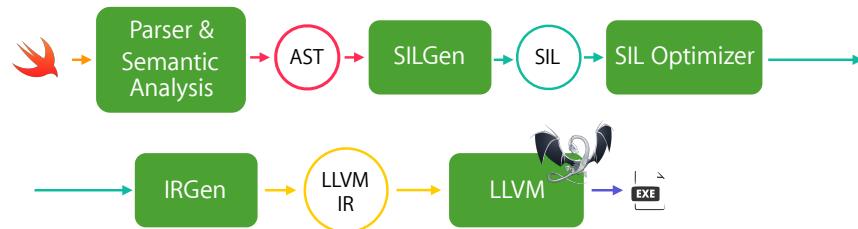


For more about SIL, see “Swift’s High-Level IR” talk:

- LLVM Developer’s Meeting 2015
- Slides & Video: <http://llvm.org/devmtg/2015-10/>

One interesting thing that will come up later is that it has a high level, language specific SSA-based IR called SIL. This enables key optimizations like generic specialization, class hierarchy analysis, etc.

# Swift Compiler Design

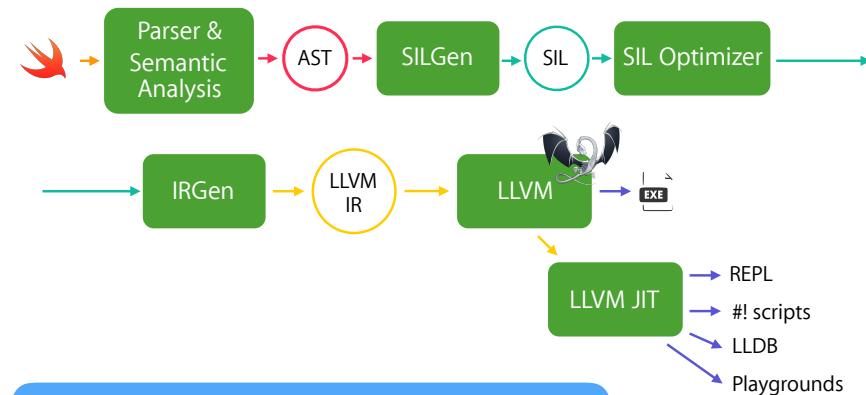


For more about SIL, see “Swift’s High-Level IR” talk:

- LLVM Developer’s Meeting 2015
- Slides & Video: <http://llvm.org/devmtg/2015-10/>

Swift compiles to LLVM IR and since it uses static compilation, it produces an executable in the end.

# Swift Compiler Design



For more about SIL, see "Swift's High-Level IR" talk:

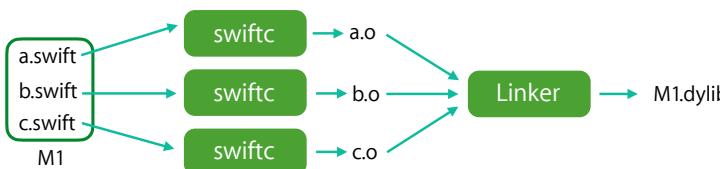
- LLVM Developer's Meeting 2015
- Slides & Video: <http://llvm.org/devmtg/2015-10/>

That said, many of the things we talked about earlier are actually powered by the LLVM JIT, including the REPL, #! scripts, Playgrounds, and the LLDB debugger. Being able to pick and choose the right compilation model for a specific use case is a strong point of LLVM.

Lets talk in more detail about how this compilation process can work out in difference configurations.

## Simple Static Compilation

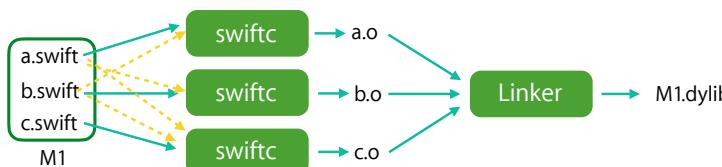
- Standard C compiler workflow:
  - Each .swift file is compiled into a .o file
  - Lazily parse other files in the module
  - Traditional linker combines them



The simplest approach is to treat Swift a lot like we treat the C compiler, where the Swift compiler is invoked once per source file, to produce an object file that is combined by a traditional linker.

## Simple Static Compilation

- Standard C compiler workflow:
  - Each .swift file is compiled into a .o file
  - Lazily parse other files in the module
  - Traditional linker combines them
- Pros: Simple, incremental compilation, good for debug builds
- Cons: Poor generated code performance



One complexity comes from the fact that files within a module can freely refer to declarations in other files in the module. This means that the compiler actually has to parse multiple files in many cases, we try to be as lazy as possible here, but there is still work to be done.

In the (absolute) worse case, this compilation approach can lead to  $N^2$  parsing and type checking, though in practice, Swift requires types on the signatures of (almost all) declarations, so the bodies of declarations in other files can be skipped (the exception is var/let declarations whose types are often inferred from their initializer expression).

We use this compilation model today for debug builds because incremental build times are relatively fast. That said, runtime performance is not very high, because any abstractions within the module are very expensive.

## Naive Whole Module Optimization

- One compiler invocation per module:
  - Parse, optimize, and code gen entire module at once
- Pros:
  - Simple
  - Enables cross-file optimization
  - Little redundant work
- Cons: No incremental compilation, doesn't use multiple cores



For release builds, one obvious improvement is to enable cross-file optimization, we call this Whole Module Optimization. This is a great step for the performance of the compiled code, but leads to unacceptable incremental build times, and if you build on a multicore machine, we don't use more than one core.

## Parallel Whole Module Optimization

- Parse and optimize entire module together:
  - LLVM optimization + codegen stages are multithreaded
  - Provides incremental compilation by hashing input IR
- Scalability:
  - Pro: Parallelizes most expensive part of compiler
  - Con: AST and SIL for entire module still in memory at once



<https://swift.org/blog/whole-module-optimizations/>

Recently we improved this by multithreading the LLVM codegen part of the pipeline. Code generation is where the  $N^2$  and NP complete algorithms live, so this gives us a huge improvement in practice. It is still a long term concern for module scalability, and it is still not incremental enough.

We'd like to optimize across multiple modules, and eventually scale to optimizing hundreds or thousands of packages all in one highly tuned application.

## Summary-based Parallel WMO?

- Incorporate concepts proven by LLVM's ThinLTO:
  - Cache interprocedural summaries for each file
  - No process needs to load SIL for entire module into memory
  - More principled partitioning of code
- Advantages:
  - Better scalability for huge modules
  - Scalability to enable cross-module optimization
  - Potential for better incremental builds

See "ThinLTO: Scalable Link Time Optimization":

- LLVM Developer's Meeting 2016
- Slides & Video: <http://llvm.org/devmtg/2016-11/>

This isn't something that we're likely to tackle in the next year or two, but an exciting related technology is the ThinLTO approach for scalable LTO growing in the LLVM community. Many of the insights and approaches can be directly applied to the Swift compiler. When we do this, we can see huge wins for release/production builds.

## What about JIT Compilation?

- Static compilation optimizes for one set of tradeoffs:
  - Startup time, memory utilization, determinism
- JITs optimize for:
  - Sustained execution speed, adaptation to phase changes
  - Could be great for Swift on the Server
- Use LLVM JIT? Integrate with other compiler frameworks?
- Hybrid static + JIT compiler approach?

I dream about the day when we can speed up the edit/compile/run cycle by using a JIT compiler to get the process running, before all the code is even compiled. In addition to speeding the development cycle, this sort of approach could provide much higher execution performance for debug builds, by using the time \*after\* the process starts up to continuously optimize the hot code.

Unfortunately, getting here requires a lot of work and has some interesting open questions when it comes to dependence tracking for changes (how you invalidate previously compiled code). That said, this would be a really phenomenal research area for someone to tackle.

# Automatic Memory Management

Lets switch gears and talk about Swift's approach to memory management. The right default for any safe language is automatic memory management.

## ARC: Automatic Reference Counting

- Each heap object carries a reference count:
  - Compiler maintains count when references are added/removed
  - Requires **atomic** increment/decrement operations
- Calling convention specifies ownership for received/returned objects:
  - Returned objects are "+1"
  - Guaranteed parameters can be passed "+0"
- ARC Optimizer:
  - Redundancy analysis hoists retain/release out of loops, etc
  - Language model reserves right to destroy objects early

For that, Swift uses an approach called ARC.

The compiler can optimize Swift ARC better than (e.g.) a C++ compiler can optimize shared pointers, because the language model explicitly allows early destruction of objects.

Overall, ARC is great. There are two principle downsides to ARC that people cite: one is the need for atomic increment/decrements, which can be slow. The second is that it requires paying attention to cycles in the heap graph.

## Object Graph Cycles Cause Leaks

```
class List {  
    var next : List?  
    var prev : List?  
}
```



To see why, lets consider a simple doubly linked list.

Doubly linked lists are not often used in practice, it is more common to see parent/child and owner/ownee relationships. Just using them in this talk to explain the issue and what Swift provides.

## Weak, Unowned and Unsafe Pointers

```
class List {  
    var next : List?  
    weak var prev : List?  
}
```

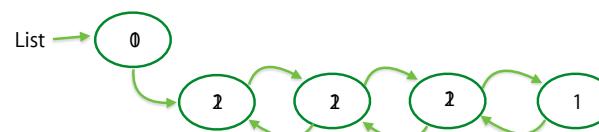


Strong references are for ownership relations, weak/unowned is for “back references”.

Weak/Unowned/Unsafe provide three levels on the safety/performance tradeoff space. Could spend 30 minutes of the talk discussing the different use cases for these three options, and why Swift has all three, but that is a topic for another day.

## Custom Cycle Breaking with 'deinit'

```
class Node {
    var next, prev : Node?
}
class List {
    var head : Node
    deinit {
        var n : Node? = head
        while let node = n {
            node.prev = nil
            n = node.next
        }
    }
}
```



Given that heap objects have deinit (aka de-initializers or destructors), other patterns are available, and the RC world has lots of them. In this example, a custom de-initializer is used to eliminate the back references, allowing the nodes to be cleanly deallocated.

An arguably more idiomatic way to write this deinit is with a for/in loop, like this:

```
deinit {
    for n in sequence(first: head, next: { $0.next }) {
        n.prev = nil
    }
}
```

## Why not a cycle collector?

- Commonly requested, technically feasible to implement:
  - Wouldn't it be great to not think or worry about cycles?

People frequently request that we implement a cycle collector, particularly when they are new to ARC. Other communities (e.g. Python) provide a cycle collector, and it is certainly technically feasible to implement, so why don't we provide one?

There are two answers:

## Why not a cycle collector?

- Commonly requested, technically feasible to implement:
  - Wouldn't it be great to not think or worry about cycles?
- Thinking a (little bit) about memory is good!
  - "Leaks" are possible with any memory management model
  - Code is read/maintained far more than it is written
  - ARC provides declarative model for memory management

The first is that we think it is important for programmers to do \*some\* thinking about their memory management approach, because you can have "leaks" in your code with any memory management approach. It is typical in garbage collected applications (and yes, ARC is susceptible to this too) for accidentally hold onto entire object graphs longer than you intend, something we call "abandoned" memory. Ownership relationships are also really important to think about, because they affect much more than memory management.

Once someone writing the code has done that thinking, we believe it is just as important for them to capture that thought into an artifact in the code that later maintainers can reason about.

Here is a way to think about the ARC model: Compared to imperative manual memory management code, it is an upgrade because it provides a declarative way to achieve the same thing. Compared to a classical GC system, these declarative annotations (like "weak") that are captured in the code serve as compiler enforced documentation about the structure of the application.

To help programmers discover the annotations they need, we focus on developer tools like leak detection and heap visualization. It would be an interesting research area to take these one step further to automatically suggest introducing weak in the right places.

## Why not a cycle collector?

- Commonly requested, technically feasible to implement:
  - Wouldn't it be great to not think or worry about cycles?
- Thinking a (little bit) about memory is good!
  - "Leaks" are possible with any memory management model
  - Code is read/maintained far more than it is written
  - ARC provides declarative model for memory management
- Availability of a cycle collector would partition the community:
  - Some packages would rely on it, some would not
  - If on by default, almost everything would rely on it

A second, and equally important reason is that the presence of a cycle collector in the Swift ecosystem would fracture the community: some packages would depend on it, and others wouldn't. This means that if you didn't want to use the cycle collector that there would be a lot of great swift packages you couldn't use.

We've seen similar dual models exist in other communities. For example, the D language community has a manual management vs GC divide. We want to learn from this experience.

Overall, we feel that introducing a cycle collector would lead us to inevitable conclusion where everyone would use it, and at that point, we might as well just use a tracing collector. Ok, since I mentioned it, why not just use a tracing collector in the first place???

## Why not a tracing GC?

- Native interoperability with unmanaged code
- Deterministic destruction provides:
  - No “finalizer problems” like resurrection, threading, etc.
  - Deterministic performance: can test/debug performance stutters

There are several reasons. The first is that interoperability with existing native C and other APIs is incredibly important to us. This can be done in tracing GC's, but depending on the design can introduce significant design complexity like pinning, JNI-like interfaces, and other performance and conceptual overhead that ARC doesn't require.

Second a major feature of ARC is that you get deterministic destruction of objects. This eliminates “finalizers” as a concept, and thus define away all the well known problems that finalizers bring with them (object resurrection, finalizers running on the wrong thread, finalizers running multiple times, ....)

ARC doesn't provide a real-time collection model (dropping the last reference to a very large object graph can take a long time to clean up), but deterministic destruction gives you the ability to test for (and debug) these situations. OTOH, with tracing collectors, they always seem to kick in at unfortunate times – and workarounds like manually running the collector at known good times introduces other problems.

## Why not a tracing GC?

- Native interoperability with unmanaged code
- Deterministic destruction provides:
  - No “finalizer problems” like resurrection, threading, etc.
  - Deterministic performance: can test/debug performance stutters
- Performance:
  - GC use ~3-4x more memory than ARC to achieve good performance
  - Memory usage is very important for mobile and cloud apps
  - Incremental/concurrent GCs slow the mutator like ARC does

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management  
Matthew Hertz, Emery D. Berger. OOPSLA'05

More generally, it is widely known that ARC uses dramatically less memory than tracing collectors do. The Herz/Berger paper is a peer reviewed survey of this effect in the context of Java, which shows that GC's need ~4x as much memory as manual management (or ARC) does to achieve the same performance. This makes intuitive sense: a GC collection needs to walk the entire heap, and if there is too little “scratch” space to play with, the collector needs to kick in much more frequently.

A paraphrase from the paper: “With heap size 4–5x of what the app uses, it runs full speed. With only three times as much memory, it runs 17% slower on average, and with twice as much memory, it runs 70% slower. Garbage collection also is more susceptible to paging when physical memory is scarce.”

Memory use arguably matters more now than it did 20 years ago, due to the rise of mobile, and the fact that many Cloud providers make you pay for memory utilization. This means that writing your cloud app in Swift, your operational costs could be much lower than a GC language.

Slow the mutator: It is commonly forgotten that GC's require write barriers, loop safe points, and other additional instructions to be inserted into the application code. Tricolor algorithms for concurrent collection require maintaining color bits in object headers, leading to overhead very similar to ARC's.

## Open Topics

- Reference count speculation
- Escape analysis to allow non-atomic operations
- Per-thread reference counts
- Your idea here!
- Short term approach:
  - Pursuing enhancements to ARC optimizer within its current model

That said, ARC is not a panacea. The performance problems it can cause are real in some important cases, and this is an area that we need to continue to put energy into.

Looking back over the last 20 years, thanks to Java, a tremendous amount of R&D has been poured into GC technologies. If only 10% of that were poured into ARC over the next 20 years, I think some major breakthroughs could happen.

That said, Swift programmers can't wait 20 years, and sometimes automatic memory management is just the wrong approach for a problem. For that, we provide manual memory management...

# Manual Memory Management

Automatic memory management is great, and is the right default. However, sometimes it gets in the way: perhaps for performance reasons, because you want precise control over where allocations happen (e.g. in a systems context), or because you want to work with a legacy API that works in terms of its own allocator.

ARC and GC are effectively “dynamic” memory management (through dynamic refcounts, mark/sweep passes), whereas you can look at most manual techniques as being static, based on program guarantees or knowledge the programmer has about the program structure.

## Unsafe Manual Memory Management

- Safe abstractions built on top of low-level unsafe APIs
- UnsafePointer APIs provides C-level pointer support:
  - Loading, storing, pointer arithmetic
  - Initialization/destruction, accessing untyped memory, etc.
  - Even malloc() and free() work out of the box
- Unmanaged<T> provides manual reference counting support:
  - Retain/release heap instances
  - Interop with older Core Foundation APIs

If ARC performance is too poor, your only  
solution is to forgo safety! 😞 🙄

Swift follows the well known approach of building safe abstractions (like Array) on top of unsafe abstractions (like UnsafeMutablePointer).

It has great support for manual memory management, both for classes (through Unmanaged) and the UnsafePointer family (for everything else)...

... but only if you give up safety! What if we gave programmers the ability to express their knowledge about the program guarantees directly in the code? If we could do that, we could achieve a static memory management model that was also safe!

## Future: Memory Ownership Support

- Opt-in linear/affine typing for references:
  - Fast: Unique ownership guarantees no ARC 🚗
  - Safe: Correctness is enforced statically 🎉
- Type system enhancements:
  - Owned: have responsibility for value
  - Borrowed: just using it temporarily
- Low burden, but still more than most users care about:
  - Aiming at users who want C++ level performance
  - Everyone else should be able to ignore it

That is exactly what our work on introducing memory ownership into Swift is focused on. We are learning from Cyclone, Rust, and other languages that have tackled linear/affine typing, but taking a Swift approach to it.

Unlike these systems (which make ownership central to the programming model) we want bring “progressive disclosure of complexity” to bear on this problem, and make it so most people can completely ignore this... until they have a performance problem or other modeling problem that benefits from it.

Assuming this works out like we think it will, this will provide the ability to achieve C++ level performance in Swift – which can open entirely new doors for applicability of Swift lower in the stack for systems programming. I wouldn’t be surprised if some (particularly performance and low level systems focused) communities even opt to \*only\* program in this mode.

## Optimizing ARC Overhead

- Annotations can be used to optimize use of copyable types

```
extension Collection {  
    func map<T>(_ f: (Element) -> T) -> [T] {  
        var result = [T]()  
        for element in self {  
            result.append(f(element))  
        }  
        return result  
    }  
}
```

These annotations allow programmers to eliminate extra work for shared references, too. Here is a simple implementation of the ‘map’ method in a protocol extension on Collection.

Functions normally take ownership of their arguments, so that call will have to copy each element, incrementing any reference counts inside it.

## Optimizing ARC Overhead

- Annotations can be used to optimize use of copyable types

```
extension Collection {  
    func map<T>(_ f: (borrowed Element) -> T) -> [T] {  
        var result = [T]()  
        for element in self {  
            result.append(f(element))  
        }  
        return result  
    }  
}
```

- Avoids temporary copy of each element
  - Eliminates retain/release of any references inside of it

By simply annotating the argument as borrowed, we can easily avoid that, letting the function decide whether it needs to copy. This annotation also makes this method correct for collections of affine types.

We expect to get a proposal draft out to the swift-evolution community later this year or early next year.

# Concurrency & Reliability

## Today's Concurrency Model

- Proven “Grand Central Dispatch” APIs for task based parallelism:

```
let backgroundQueue = DispatchQueue(label: "com.myApp.queue")
...
backgroundQueue.async {
    print("Dispatched to background queue")
```

- No language level concurrency primitives
- No memory consistency model
- No language support for asynchronous calls

GCD APIs are available cross platform, and provide a great basic way to model concurrency in an application. Current Swift language doesn't have any concurrency language/compiler features in it.

## Problems / Opportunities to Improve

- Completion closures lead to async “pyramid of doom”:

```
backgroundQueue.async {
    let work = doWork()
    readFileAsync(filename) { contents in
        let result = processFile(work, contents)
        DispatchQueue.main.async {
            updateUI(result)
        }
    }
}
```

- Prevalent shared mutable state ⇒ concurrency bugs
- Difficult to know what state/data is owned by what task
- Foundation level async APIs are inconsistent

Difficult to know: unless there are comments, it is difficult to know what data is “owned” by a particular dispatch queue. It is all described imperatively in code, there is no declarative structure.

## Swift 5+: Tackling Task Concurrency

- Goal: start discussions in Spring/Summer 2017:
  - Aim to have a “manifesto” design sketch by Fall 2017
  - First deliverables could arrive in Swift 5 (2018)
- Public design using swift-evolution or workgroup model
- Aggressive goal means focusing, likely can’t tackle:
  - Memory consistency model
  - Data parallelism
  - ...

Planning for future releases is inherently fraught with peril, since we don’t know what will happen in this release, and we don’t know what help the open source community will provide. That said, we would like to start work on task based concurrency with a goal of at least the first pieces of it hitting Swift 5.

## One Possible Concurrency Direction\*

- async/await for elegant async operations:
  - Built on coroutines: useful for generators etc
  - Proven in C#, Javascript, Python. Coming to Kotlin and others
  - Solves completion handler pyramid of doom
- Actor model to define tasks, along with the state managed by it:

```
actor NetworkRequestHandler {  
    var localState : UserID  
  
    async func processRequest(connection : Connection) {  
        ...  
    }  
}
```

\* Just ideas to kindle discussion, plan is not endorsed by the core team

Async/await are effectively proven at this point and would fit well with the Swift type system and structure, we should probably just do it.

Erlang/Akka have proven actor models. Swift providing syntax yields a declarative model which enables programmers to reason about what mutable state goes with each task. Each actor is effectively a DispatchQueue + state it manages + operations that act on it.

Actor approach also generalizes well to future. It could be a great way for handling heterogenous compute (e.g. GPU tasks), distributed compute, etc since each async “call” to an actor really is a message send (thus could be DMA or IPC).

## Today's Reliability Model

- Static type system:
  - Designed to encourage thought about unusual cases
  - Optionals, error handling model, default in switches, etc
- Dynamic checks catch the rest:
  - Array bounds checks
  - Integer overflow checks
  - Force unwrapping optional
  - ...

A failed dynamic check  
**terminates entire process**



For the purposes of this talk, Reliability means: after you write your swift app, does it work, or do you have to spend lots of time chasing endless bugs in it?

Good defaults define away a ton of reliability problems, and many people have been happy with how stable their swift apps are, even on their first try. This is because Swift turns many common problems into compile time errors. Another second class of common errors are caught at runtime, and Swift has a "fail fast" approach to this, which often makes it easy to reason about the cause of a problem.

That said, terminating a process outright when a dynamic check is unfortunate for many use-cases:

- UI apps want to be able to save document recovery state
- One problematic task on a server shouldn't bring down the entire server, just the current job

## Possible Future Reliability Model?

- Terminate failing Actor instead of entire process:
  - All “awaits” on an actor’s method throw an error
    - Enables custom failure recovery
  - Runtime cleans up resources owned by that Actor
- Hard questions:
  - Is cleanup guaranteed or best effort?
  - How does cleanup work? How much space cost for metadata?
  - Are deinit’s run or not? New kind of deinit?
  - Mutable shared state (if allowed) could be left inconsistent!

Borrowing another page from Erlang, we can build on top of the actor model to provide a solution for this. Instead of taking down the entire process, only take down the current actor.

Deinit's are problematic because the local state within the actor is potentially inconsistent.

## Swift Language Evolution

As I mentioned, Swift is moving fast, and fully open source. This includes the design of the language itself.

## Swift Evolution Process

- Socialize change on mailing list
- Proposal submitted as a pull request
- Pull request accepted to start review
- Formal review on mailing lists
- Core team arbitrates a decision

<https://github.com/apple/swift-evolution>

The screenshot shows a GitHub pull request page for proposal SE-0046. The title of the pull request is "Establish consistent label behavior across parameters including first labels". The proposal has 111 lines (82 sloc) and is 5.94 KB in size. It was created by Jake Carter and Erica Sadun, accepted by Jake Carter, and managed by Chris Lattner. The introduction section discusses the goal of normalizing first parameter declarations. The motivation section explains the current state of Swift 2 where internal implementation names use different symbols than external consumption names, and how this proposal aims to统一 them. A code snippet example is shown:

```
func foo(a: T, b: U, c: V)
```

We follow a standardized process and have over a hundred formal proposals over the last year, most of which were eventually accepted (sometimes after revision).

## Active Release: Swift 4 in Late 2017

- Split into Stage 1 and Stage 2 goals
- Stage 1: Prioritize critical user needs and ABI stability
  - Improved compile time, compiler reliability, and error messages
  - New generics features needed by standard library
  - Standard library API improvements, e.g. String
  - Calling conventions, runtime layout, name mangling, ...
- Stage 2:
  - Re-evaluate release in early Spring
  - Scope in new improvements based on time remaining
- Feature complete in May, ship in Late 2017

Major problem we have with Swift is that there are so many things we “could” do, we need to be disciplined about our approach to managing releases.

For the Swift 4 cycle we are trying to learn from some of the problems we had in the Swift 3 cycle – in order to hit important goals for the community, we are trying to keep things more focused on those things. That said, we still want to enable some organic work to happen in “stage 2”.

## **Swift: Challenges and Opportunities**

- Swift is still young, and moving fast
- Design presents interesting new sets of tradeoffs
- Fertile ground for research & development
- Great time to get involved!



Swift.org