How Swift Achieved Dynamic Linking Where Rust Couldn't

Aria Desires

November 7th, 2019

For those who don't follow Swift's development, ABI stability has been one of its most ambitious projects and possibly it's defining feature, and it finally shipped in Swift 5. The result is something I find endlessly fascinating, because I think Swift has pushed the notion of ABI stability farther than any language without much compromise.

So I decided to write up a bunch of the interesting high-level details of Swift's ABI. This **is not** a complete reference for Swift's ABI, but rather an abstract look at its implementation strategy. If you really want to know exactly how it allocates registers or mangles names, look somewhere else.

Also for context on why I'm writing this, I'm just naturally inclined to compare the design of Swift to Rust, because those are the two languages I have helped develop. Also some folks like to complain that Rust doesn't bother with ABI stability, and I think looking at how Swift *does* helps elucidate why that is.

This article is broken up into two sections: background and details. Feel free to skip to the details if you're very comfortable with the problems inherent to producing a robust dynamically linked system interface.

If you aren't comfortable with the basic concepts of type layouts, ABIs, and calling conventions, I recommend reading the article I wrote on the basic concepts of type layout and ABI as they pertain to Rust.

Also huge thanks to the Swift devs for answering all of the questions I had and correcting my misunderstandings!

§1 Background

§1.1 Swift TLDR

I know a lot of people don't really follow Swift, and it can be hard to understand what they've really accomplished without some context of what the language is like, so here's a TL;DR of the language's shape:

- Exists to replace Objective-C on Apple's platforms, oriented at application development
  - natively interoperates with Objective-C
  - has actual classes and inheritance
- At a distance, very similar to Rust (but "higher-level")
  - interfaces, generics, closures, enums with payloads, unsafe escape hatch
  - no lifetimes; Automatic Reference Counting (ARC) used for complex cases
  - simple function-scoped mutable borrows (inout)
  - Ahead-Of-Time (AOT) compiled
- An emphasis on "value semantics"
  - structs/primitives ("values") are "mutable xor shared", stored inline
  - collections implement value semantics by being Copy-On-Write (CoW) (using ARC)
  - classes are mutably shared and boxed (using ARC), undermining value semantics (can even cause data races)
- An emphasis on things Just Working
  - language may freely allocate to make things Work

- generic code may be polymorphically compiled
- fields may secretly be getter-setter pairs
- ARC and CoW can easily result in surprising performance cliffs
- tons of overloading and syntactic sugar

Don't worry about fully understanding all of these, we'll dig into the really important ones and their implications as we go on.

§1.2 What Is ABI Stability and Dynamic Linking

When the Swift developers talk about "ABI Stability" they have exactly one thing in mind: they want native system APIs for MacOS and iOS to be written in Swift, and for you to dynamically link to them. This includes dynamically linking to a single system-wide copy of the Swift Standard Library.

Ok so what's dynamic linking? For our purposes it's a system where you can compile an application against some abstract *description* of an interface without providing an actual implementation of it. This produces an application that on its own will not work properly, as part of its implementation is missing.

To run properly, it must tell the system's *dynamic linker* about all of the interfaces it needs implementations for, which we call *dynamic libraries* (dylibs). Assuming everything goes right, those implementations get hooked up to the application and everything Just Works.

Dynamic linking is very important to system APIs because it's what allows the system's implementation to be updated without also rebuilding all the applications that run on it. The applications don't care about what implementation they get, as long as it conforms to the interface they were built against.

It can also significantly reduce a system's memory footprint by making every application share the same implementation of a library (Apple cares about this a lot on its mobile devices).

Since Swift is AOT compiled, the application and the dylib both have to make a bunch of assumptions on how to communicate with the other side long before they're linked together. These assumptions are what we call ABI (an Application's *Binary* Interface), and since it needs to be consistent over a long period of time, that ABI better be stable.

So dynamic linking is our goal, and ABI stability is just a means to that end.

For our purposes, an ABI can be regarded as 3 things:

1. The layout of types
2. The calling convention of functions
3. The names of symbols

If you can define these details and never break them, you have a stable ABI, and dynamic linking can be performed. (Ignoring trivial cases where both the dylib and application were built together and ABI stability is irrelevant.)

Now to be clear, ABI stability isn't technically a property of a programming language. It's really a property of a system and its toolchain. To understand this, let's look at history's greatest champion of ABI stability and dynamic linking: C.

All the major OSes make use of C for their dynamically linked system APIs. From this we can conclude that C "has" a stable ABI. But here's the catch: if you compile some C code for dynamic linking on Ubuntu, that compiled artifact won't work on MacOS or Windows. Heck, even if you compile it for 64-bit Windows it won't work on 32-bit Windows!

Why? Because ABI is something defined by the *platform*. It's not even something that necessarily needs to be documented. The platform vendor can just require you to use a particular compiler toolchain that happens to implement their stable ABI.

(As it turns out, this is actually the reality of Swift's Stabilized ABIs on Apple platforms. They're not actually properly documented, xcode just implements it and the devs will do their best not to break it. They're not opposed to documenting it, it's just a lot of work and shipping was understandably higher-priority. Thankfully I don't really care about the details, or the difference between the ABIs on MacOS and iOS, or implementations other than Apple's, so I can keep saying "Swift's ABI" and it won't be a problem.)

But if that's the case, why don't platform vendors provide stable ABIs for lots of other languages? Well it turns out that the language isn't completely irrelevant here. Although ABI isn't "part" of C itself, it *is* relatively friendly to the concept. Many other languages aren't.

To understand why C is friendly to ABI stability, let's look at its much less friendly big brother, C++.

Templated C++ functions cannot have their implementations dynamically linked. If I provide you with a system header that provides the following declaration, you simply can't use it:

```
template <typename T>
bool process(T value);
```

This is because *it has no symbol*. C++ templates are monomorphically compiled, which is a fancy way of saying that the way to use them is to copy-paste the implementation with all the templates replaced with a particular value.

So if I want to call `process<int>(0)`, I need to have the implementation available to copy-paste it with `int` replacing `T`. Needing to have the implementation available at compile-time completely undermines the concept of dynamic linking.

Now perhaps the platform could make a promise that it has precompiled several monomorphic instances, so say symbols for `process<int>` and `process<bool>` are available. You could make that work, but then the function wouldn't really be meaningfully templated anymore, as only those two explicitly blessed substitutions would be valid.

There would be little difference from simply providing a header containing:

```
bool process(int value);
bool process(bool value);
```

Now a header *could* just include the template's implementation, but what that would really be guaranteeing is that that particular implementation will *always* be valid. Future versions of the header could introduce new implementations, but a robust system would have to assume applications could using either, or perhaps even both at the same time.

This is no different from a C macro or `inline` function, but I think it's fair to say that templates are a little more important in C++.

For comparison, most platforms provide a dynamically linked version of the C standard library, and everyone uses it. On the other hand, C++'s standard library isn't very useful to dynamically link to; it's literally called the Standard *Template* Library!

In spite of this issue (and many others), C++ *can* be dynamically linked and used in an ABI-stable way! It's just that it ends up looking a lot more like a C interface due to the limitations.

Idiomatic Rust is similarly hostile to dynamic linking (it also uses monomorphization), and so an ABI-stable Rust would also end up only really supporting C-like interfaces. Rust has largely just embraced that fact, focusing its attention on other concerns.

§1.3 Swift's Stable ABI

I have now made some seemingly contradictory claims:

- Swift has similar features to Rust
- Rust's features make it hostile to dynamic linking
- Swift is great at dynamic linking

The secret lies in where the two languages diverge: dynamism. Rust is a *very* static and explicit language, reflecting the sensibilities of its developers and early adopters. Swift's developers preferred a much more dynamic and implicit design, and so that's what they made.

As it turns out, hiding implementation details and doing more work at runtime is *really* friendly to dynamic linking. Who'd've thought dynamic linking was dynamic?

But what's really interesting about Swift is the ways it's *not* dynamic.

It's actually fairly trivial to dynamically link a system where all the implementation details are hidden behind uniformity and dynamism. In the extreme case, we could make a system where everything is an opaque pointer and there's only one function that just sends things strings containing commands. Such a system would have a very simple ABI!

And indeed, in the 90's there was a big push in this direction with Microsoft embracing COM and Apple embracing Objective-C as ways to build system interfaces with simple and robust ABIs.

But Swift didn't do this. Swift tries its hardest to generate code comparable to what you would expect from Rust or C++, and how it accomplishes that is what makes its ABI so interesting.

It's worth noting that the Swift devs disagree with the Rust and C++ codegen orthodoxy in one major way: they care much more about code sizes (as in the amount of executable code produced). More specifically, they care a lot more about making efficient usage of the cpu's instruction cache, because they believe it's better for system-wide power usage. Apple championing this concern makes a lot of sense, given their suite of battery-powered devices.

It's harder for third party developers to care about this, as they will naturally only control some small part of the software running on a device, and typical benchmarking strategies don't really capture "this change made your application run faster but is making some background services less responsive and hurting battery life". Hence C++ and Rust inevitably pushing towards "more code, more fast".

This is all to say that some things which seem like compromises made for ABI stability's sake are genuinely just regarded as desirable.

I never got any great concrete numbers on this concern from the Swift or Foundation folks, would definitely love to see some! *Waves at the Apple employees reading this*.

§1.4 Resilience and Library Evolution

The Swift developers cover this topic fairly well in their documentation. I'll just be giving a simplified version, focusing on the basic motivation.

Resilience is the core concept behind Swift's dynamic linking story. It means that things default to having ABIs that are *resilient* to breaking when the implementation changes in an API-preserving way (nothing can save API-breaking changes). This allows developers to create dynamically linked and idiomatic-feeling libraries that can still easily evolve their implementations.

This is in contrast to C, which only makes it *possible* to create a stable ABI with proper vigilance and foresight. This is because C requires you to commit to many of the ABI details of your interface upfront, even if you're uncertain about them. If you don't want to commit to those details, you'll have to change the shape of your API to hide them.

When compiled as a dylib, Swift defaults to implicitly hiding as many details as possible, requiring you to opt into guarantees by adding annotations. Crucially, these annotations don't affect the shape of an API, they're "only" for optimizing the ABI, at the cost of resilience.

Additionally, *some* ABI annotations can be added after a library has been published without breaking the old ABI. Applications compiled against new annotations are able to use that information to run faster, at the cost of compatibility with older versions of the library.

(It seems the Swift devs ran out of time/resources and quite reasonably cut a few corners in this department. Several annotations which plausibly could be done in a backwards-compatible way are ultimately breaking to add. Ah well, pobody's nerfect.)

This is all very abstract, let's look at a simple library evolution example.

Let's say we draft up a simple FileMetadata interface in C:

```
// version 1
typedef struct {
```

```
    int64_t size;
} FileMetadata;

bool get_file_metadata(char* path, FileMetadata* output);
```

Which would be called as:

```
FileMetadata metadata;
if (!get_file_metadata("/my/sweet/file.txt", &metadata) {
    printf("error!");
    return;
}
printf("file size %lld", metadata.size);
```

Now let's say we realize that this function should also provide info on when it was last modified:

```
// version 2
typedef struct {
    int64_t last_modified_time; // 64 bits CLEARLY enough...
    int64_t size;
} FileMetadata;

bool get_file_metadata(char* path, FileMetadata* output);
```

Oops, we've messed up our ABI! Our hypothetical caller is stack allocating a FileMetadata, so they're assuming it has a particular size and alignment. Additionally, they're directly accessing the `size` field, which they assume is at a particular offset in the struct.

Both of those assumptions were violated by our change. This didn't necessarily have to happen. There's a few common approaches we could have taken to allow for this change. For instance we could have:

- Reserved space in our struct for future use
- Made FileMetadata an opaque type, requiring function calls to get the fields
- Given FileMetadata a pointer to its "version 2" data (opaque in "version 1")

Unfortunately, all of these require us to have the foresight to do them while also changing the way users make use of our API. In some sense, the API becomes less "idiomatic" to accommodate future changes. Additionally, we will forever be burdened with this complexity even if we eventually determine that the API is complete enough to guarantee its details.

Swift doesn't require you to make this compromise.

The following two designs are totally ABI compatible while remaining perfectly idiomatic to use:

```
// version 1
public struct FileMetadata {
    public var size: Int64
}

public func getFileMetadata(_ path: String) -> FileMetadata?
```

```
// version 2
public struct FileMetadata {
    public var lastModifiedTime: Int64 // just add this field, that's it
    public var size: Int64
}

public func getFileMetadata(_ path: String) -> FileMetadata?
```

Unfortunately, guaranteeing the layout of FileMetadata using the `@frozen` attribute in future versions *would* be an ABI breaking change under the current design. Hopefully it will be clear why by the end of this document.

§2 Details

Ok! Now for the details, where I will in fact be ignoring the *actual* details and instead discussing the high level ideas behind them.

Once again, feel free to check out Swift's documentation of the annotations that are used to manage abi resilience. That covers a lot of motivation and the fine-grain details of what you can and can't do.

§2.1 Resilient Type Layout

By default, a type that is defined by a dylib has a *resilient* layout. This means that the size, alignment, stride, and extra inhabitants of that type aren't statically known to the application. To get that information, it must ask the dylib for that type's *value witness table* at runtime.

"Witness tables" are Swift's term for what are ultimately vtables. The details of how these tables are acquired and laid out don't really interest me, so I won't discuss that.

> Ok actually it is Interesting that Swift needs to be able to generate witness tables at runtime to deal with the fact that generic type substitutions can't be statically predicted in the face of dynamic linking of generic code, but that's getting way ahead of ourselves.

The *value* witness table is just the "vtable of basic stuff you might want to know about any type", much like how Java's `Object` type is used. So it has all the stuff like size, alignment, stride, extra inhabitants, move/copy constructors (for ARC), and destructors.

At this point those with experience in language design probably suspect this results in resilient types having to be boxed and passed around as a pointer. And those suspicions are indeed correct... but not quite.

See what's really interesting about resilient layout is that it's only something that the application is forced to deal with, and only in a very limited way. Inside the boundaries of the dylib where all of its own implementation details are statically known, the type is handled as if it wasn't resilient.

Inside the dylib a resilient struct is stored inline, stored on the stack, passed around by value, and even scalarized. But once we move outside the dylib something else must be done.

We could potentially accomplish this with expensive type layout conversion at the boundaries, but we don't! Type layouts are always the same on both sides of the resilience boundary!

Type layouts are always the same on both sides of the resilience boundary!?!?

Yes!

The key insight here is that laying out things inline can actually be done dynamically with relative ease. Memory allocators and pointers don't care about static layouts, they just work with completely untyped sizes, alignments, and offsets. So as long as you have all the relevant value witness tables, everything works basically fine, just with more dynamic values than usual.

The real major problem is stack allocations. llvm really doesn't like dynamic stack allocations. Yes, alloca does exist, but it's a bit messy. I believe the Swift devs managed to get it working all the time for resilient layout, but not for some of its cousins we'll see in the next section. In the general case, local variables need to actually be boxed up onto the heap. For convenience, I'll just generically refer to these dynamic stack allocations as "boxed".

Crucially this boxing this doesn't change layouts, just where local variables are stored and how they're passed in the calling convention (more on that later). Also, once there is *some* indirection everything is still stored inline. So types which already come with indirection like `Array<MyResilientStruct>` or `MyResilientClass` require *no* additional allocation, and consequently no ABI changes.

I've left out some key details, but let's address them while looking at polymorphic generics, since it turns out those are quite similar, but also more interesting!

§2.2 Polymorphic Generics

Unlike Rust and C++ which must monomorphize (copy+paste) implementations for each generic/template substitution, Swift is able to compile a generic function into a single implementation that can handle every substitution dynamically.

This has several benefits:

- Massively reducing code size
- Massively reducing the amount of code that must be compiled
- Allowing generic code to be dynamically linked

A polymorphic implementation can't be inlined or optimized as well as a monomorphic one (without a JIT), so the Swift compiler still monomorphizes things when it's possible and seems profitable. But we're making a dylib, so it's not possible for our public API.

As it turns out, polymorphically compiled generic code is really quite similar to code that handles resilient types. In both cases the basic value-witnessy properties of the type aren't statically known, and so stack values need boxing. The generic code just needs to be able to find the generic type's protocol implementations too. We can get that from the type's *protocol witness tables* which can be acquired using the same machinery we use for the value witness tables.

So really this is basically the same problem!

 Brief Aside About Existentials 

The resilient/polymorphic type machinery solves a big chunk of the Object Safety problem that heavily limits Rust's trait objects. Swift calls these Protocols as Types or just *existentials*, depending on who you ask. Generic code actually having symbols means there's no problem with it being stuffed in a vtable. Resilient layout eliminates the problems that come with dynamic "by value" manipulation of Self and any of its associated types.

Existentials are the really tricky case for stack allocations, because they can prevent the caller from knowing the size of the return value before making the call, and that really messes up alloca. So once existentials get involved, alloca goes out the window and actual boxing needs to happen.

Also associated types in function signatures still prevent existentials from being created because that creates fundamental type system problems unrelated to ABI. Every instance of MyProtocol could have a different associated type, and you can't let them get mixed up. No I'm not going to get into how Swift could use path-dependent types to deal with this.

Associated types are fine for normal polymorphic code, since generics enforce that every instance has the *same* type, which is the only issue with them in existentials.

Now, what does the presence of resilient/polymorphic types do to calling conventions?

Well first off, we have the witness tables. In the resilient case all the types are statically known, and so the implementation theoretically has all the information it needs to look up witness tables for itself. Polymorphic code has no such luxury.

Polymorphic code needs to work with any type, and structs don't contain any identifying runtime information. Worse yet, polymorphic code can be called without providing any values of that type! So the *caller* needs to pass in type information. Abstractly, we *could* pass in minimal information and have the polymorphic code look up all the witness tables, but that's really wasteful (consider calling the function in a loop). So instead Swift's actual implementation has the caller pass in pointers to every required witness table as extra arguments (automatically handled by the compiler).

With the witness tables handled, we just have the problem of passing/returning actual values. The main thing is that storing these kinds of values in registers is totally off limits; we really need to pass them around as pointers. That's sometimes a bit slower, but not a terribly huge deal given we're already signed up for dynamic linking and polymorphic compilation.

But to really understand passing these values, we need to talk about *reabstraction*.

§2.3 Reabstraction

Resilient compilation forces us to use a particular calling convention that's different from what we would use statically. For instance, in the x64 SysV ABI, the following would have all of its fields passed in registers:

```
struct Vec4 { int32_t x; int32_t y; int32_t z; int32_t w; }

void process(Vec4 vector);
```

If Vec4 were resilient, it would instead have to be passed by-reference. But remember, not all code that works with a type *needs* to handle it resiliently. For instance, if a dynamic library defines Vec4 resiliently, it should ideally still be able to handle it non-resiliently inside of itself.

Similarly, the polymorphicness (polymorphicity?) of things can be changed by their context. Consider the following Swift code that manipulates a closure:

```
// closure is very generic
func map<T, U>(_ input: T, with mapper: (T) -> U) -> U {
    return mapper(input)
}

// closure is kinda generic
func mapInt<U>(_ input: Int, with mapper: (Int) -> U) -> U {
    return map(input, with: mapper) // just delegate to generic map
}

// actual closure isn't generic
let result = mapInt(3, with: { return $0 >= 5 })
```

Assuming `map` and `mapInt` are compiled polymorphically, the closure has 3 potentially different ABIs:

- (Int) -> Bool
- (Int) -> U
- (T) -> U

The `(Int) -> Bool` ABI can potentially be ignored and discarded because we know we're passing it to something that expects `(Int) -> U`, but the `(T) -> U` ABI is completely hidden from us!

This naively results in an unfortunate conclusion: closures (and function pointers) must have the maximally generic ABI just in case that's needed. Thankfully, this conclusion is incorrect.

Instead Swift uses *reabstraction thunks*. These thunks simply wrap a function with the wrong ABI in a function with the right one. So what the compiler "actually" generates is more like this:

(note: not real Swift code because you can't explicitly talk about generics/conventions in this way)

```
// closure is very generic
func map<T, U>(_ input: T, with mapper: (T) -> U) -> U {
    return mapper(input)
}

// closure is kinda generic
func mapInt<U>(_ input: Int, with mapper: (Int) -> U) -> U {
    // reabstract it!
```

```
    let thunk: <T,U>(T) -> U = { return mapper($0) }

    return map(input, with: thunk) // just delegate to generic map
}

// actual closure isn't generic
let temp_closure: (Int) -> Bool = { return $0 >= 5 }
// reabstract it!
let thunk: <U>(Int) -> U = { return temp_closure($0) }

let result = mapInt(3, with: thunk);
```

In this way everything can use the best possible calling convention while still allowing for more generic ones to be used in different contexts.

Even without this closure passing problem, reabstraction also allows a single implementation to be used in several different contexts without having to compile different versions of it. So for instance we can reabstract a concrete protocol implementation into a polymorphic one by just wrapping all the functions in reabstraction thunks. A nice code size win!

(I believe the Swift devs don't technically call that one Reabstraction but it's close enough that I'm happy to conflate the concepts. Thunk away ABI complexities!)

Now that we have a basic idea of how resilience and polymorphism affects calling conventions, it should hopefully be clear why it's an ABI-breaking change to mark a type as `@frozen`, removing its resilient layout: it would change the way the type is passed to functions.

This could have potentially been "fixed" by making resilience part of the name mangle and providing both the resilient and non-resilient versions, but that requires robust versioning info for every attribute and could lead to a huge combinatoric explosion in the number of symbols a dylib provides. Not necessarily a great idea.

§2.4 Materialization

Resilient layout could be generalized to provide the offsets to a resilient type's public fields, but Swift actually takes this to another level: public fields don't actually need to exist by default.

Resiliently exposed fields are only exposed as getters and setters!

Getters and setters are actually a first class feature of Swift that can be used explicitly, but for resilience the compiler will implicitly introduce those getters and setters just to hide the fact that the fields physically exist, in case you change your mind.

A computed field in Swift is manipulated in exactly the same way as a real one, and so even without resilience library authors are free to replace physical fields with computed ones without changing their A**P**I. Mandating computed access just makes it A**B**I stable as well.

But here's the catch: you can take a mutable reference (inout) to a field. Even a computed one!

Swift let's you take references to things that don't exist.

The secret to this is *materialization*.

Inouts can only appear as arguments to a function, and so they're naturally scoped to a function call. As such, we can "take a reference" to a computed field by using a temporary with cleanup. So this code:

```
struct MyStruct {
    public var myField: FieldTy
}

func doTheThing(x: inout FieldTy)

var myVal = MyStruct(..)
doTheThing(&myVal.myField)
```

is (very roughly) compiled to:

```
var myVal = MyStruct(..)

var temp: FieldTy = myVal.get_myField();
doTheThing(&temp)
myVal.set_myField(temp)
```

Or to break that into steps:

1. Allocate a temporary local variable (temp)
2. Initialize temp with the getter
3. Call the inout-using function with the address of temp
4. Feed the value of temp into the setter

This creates a very interesting difference from references in Rust and C++: if you take a reference to a field, it may point to a temporary that's only valid for the scope of the call it was passed to!

Another interesting consequence of this is that Swift's mutable references all actually have a finalizer which must be executed for a write to "stick". This means they cannot be returned or

stored, as the finalizer would be lost and the referent would be deallocated.

This in turn creates a hilarious footgun many Swift developers run into where they think they're clever and convert an inout into a raw pointer (using withUnsafeMutablePointer) that they store for later and – oops it's dangling!

However you can overcome this limitation with callbacks, as follows:

```
func doSomeWork(_ callback: (inout Float) -> ()) {
    var vec = Vec4()
    // do lots of work...

    // "return" a mutable reference to the caller
    callback(&vec.x)
    // setter potentially called here to commit the writes

    // maybe do some more work...
}


doSomeWork { (val: inout Float) -> ()
    val *= 2;
}
```

Callbacks are of course very annoying and noisy, and so this was solved with the slayer of callbacks, coroutines! The same code can be rewritten as:

```
// No idea if this is valid Swift syntax, and I don't care!

func doSomeWork() -> inout Int {
    var vec = Vec4()
    // do lots of work...

    // "return" a mutable reference to the caller
    yield &vec.x
    // setter potentially called here to commit the writes

    // maybe do some more work...
}

doSomeWork() *= 2
```

Nifty!

§2.5 Ownership

Swift makes extensive use of reference counting, and as it turns out it's really expensive to constantly modify the count! Especially when you make all your collections copy-on-write (CoW), so an errant reference count bump can make an O(n) algorithm $O(n^2)$! (I actually consider this a

correctness error in the case of data structures and algorithms, but reasonable people may disagree.)

To help deal with this, Swift made ownership of reference-counted values (~classes) part of its calling convention. The most important aspect is "+0" vs "+1", referring to how the caller should change the refcount:

- +1: callee has an "owned" value it's responsible for releasing (if it doesn't escape)
- +0: callee has a "borrowed" value it's responsible for retaining (if it escapes)

Since we're talking about ownership, I'm legally required to compare this system to Rust, and the comparison is pretty straight-forward:

- +1 is a move (`T`)
- +0 is a shared immutable reference (`&T`)
- inout is a mutable exclusive reference (`&mut T`)

But there's a few key differences, due to ARC:

First and foremost, classes break "shared xor mutable" reasoning, and are effectively like using `Arc<UnsafeCell<T>>` in Rust. This is why Swift's collections provide CoW semantics, which Rust's Arc also safely provides. Yes, you can indeed trivially introduce Data Races into Swift code with classes.

Second, all Swift types can always be implicitly cloned, so a +0 can always be upgraded to an owned value without ceremony. That said, cloning in Swift (they just call it copying) is always just bumping reference counts. Other non-trivial operations, like copying an array's buffer to a new allocation, are only performed by mutations that trigger CoW. Note also that this means that if you trigger CoW on an array of arrays, you will get two independent outer arrays that still point to the same inner arrays (which are now primed to CoW if either side mutates them).

Third, +0 isn't strictly bound to pass-by-reference, and can just be a trivial bitwise copy of the value. Unfortunately, weak references require non-trivial moves because their locations are tracked for auto-nulling, so those *are* passed by reference when using +0 to keep it cheap. Yes, Swift has both copy and move constructors, although they're currently entirely ARC and not user-defined. Swift *also* has unowned references which are the same as Rust's Weak references, which have trivial moves.

In Swift's current design, +0/+1 is mostly just something the compiler does internally to optimize different calling conventions, but I think more explicit annotations are theoretically on the road map.

There's also a special path for field materialization, "modify", which returns an inout. This handles the fact that getters are naively +1, which is especially nasty for nested array operations like `array[0][2] *= 2`, as they would always trigger a huge temporary copy of the inner array!

And indeed, the subscript operator of Array contains a modify implementation:

```
 _modify {
  _makeMutableAndUnique()
  _checkSubscript_native(index)
  let address = _buffer.subscriptBaseAddress + index
  yield &address.pointee
}
```

(Interestingly, this implementation is marked as `inlineable`, and so it's actually guaranteed to always work. Array's ABI details were pretty aggressively guaranteed since it's a *relatively* simple fundamental type whose performance matters a lot.)

There's also a read-only version of modify, "read", which provides a +0 getter. That one isn't as strongly motivated, but hey it's a nice little optimization to avoid a needless retain+release.

§2.6 Opting Out of Resilience

Resilience is nifty but it clearly comes with some performance overhead. So of course Swift also provides special attributes to opt out of resilience. I had originally intended to write a whole bunch about this but it's actually really complex and subtle, so I really need to punt to Swift's actual docs.

TL;DR you can mark things as having a frozen (non-resilient) layout, exhaustively matchable, inlineable, non-subclassable, non-escaping, and a bunch of other whacky stuff which variously affects API and ABI in subtle ways.

§2.7 Esoterica

This is already ~~17~~ ~~18~~ ~~19~~ ~~20~~ 21 pages and it was supposed to just be a warmup for something else I need to do and I NEED TO STOP SO BULLET POINTS:

- Swift reserves a callee-preserved register for a method's `self` argument (pointer) to make repeated calls faster. Cool?

- Swift's exceptions (which are implemented more like Result and less like unwinding) have the error type always boxed. The caller initializes the "swift error" register to 0, and if there's

an exception the callee sets that register to hold the boxed error's pointer. This makes error propagation really fast (just don't change the register), and also doesn't require a Result to actually be materialized in the success case (avoid a ton of copies). Sadly this doesn't transfer over to Rust well, so they can't easily use the native Swift machinery that was added to llvm (swiftcc).

- Swift's implicit boxes don't actually have to be pointers, I think? They can pack small values in there and just set a high bit to distinguish an `Int32` 0 from null or something? They spent a lot of time messing around with making really-not-pointer-sized things pointer-sized with whacky hacks, and also messed around with boxes being 3 pointers wide to make the "small" box trick happen more, so I can never keep reality straight here.

- Swift used to have copy-array-of-self and move-array-of-self entries in the value witness table to deal with the "calling a bunch of copy/move constructors which might do nothing interesting" problem that C++ has, but this was replaced by just adding a flag to the witness table for whether the type is trivial for copies/moves. Less bloaty.

- Swift decouples size from stride to distinguish trailing padding from actual content. Trailing padding shows up in the stride so that arrays properly align their contents. Trailing padding doesn't show up in size so that things like enum tags and neighboring fields can use that space. It's really neat!

- Swift actually does insert a bunch of magical layout conversions when bridging between Swift and Objective-C. Initially this bridging was lazy in the case of collections, and that was super whacky. Specifically if you had an `Array<MySwiftType>` it might have been an `NSArray` containing opaque pointers, with type conversion was performed when you accessed the element. They later moved to a more eager model because this was really nasty and expensive for heavily-accessed collections. Also object-identity was messy to deal with.

- The main piece of complexity in the system for getting witness tables is that it provides a runtime reflection system for spidering through the type metadata of generic types and their associated types because you need to be able to get their witnesses too.

- I think Swift technically stabilized the set of enum tag packing optimizations it can perform, but no clue what those are. Sorry eddyb.

- Zero cost abstractions Really Aren't if they ever need to be compiled polymorphically, watch out!

- Pure Trivia: Rust actually originally tried a polymorphic design similar to Swift's, but they eventually backed off from it once the difficulties became clear. Supporting both polymorphic *and* monomorphic compilation helped Swift a lot, but I think the key difference was ultimately just that Apple had a more significant motivation than Mozilla to pursue dynamic linking and *way* more resources to throw at this very hard problem.

collapses