orphan:

Resilience

> **Warning**
>
> This is a very early design document discussing the feature of Resilience. It should not be taken as a plan of record.

Introduction

One of Swift's primary design goals is to allow efficient execution of code without sacrificing load-time abstraction of implementation.

Abstraction of implementation means that code correctly written against a published interface will correctly function when the underlying implementation changes to anything which still satisfies the original interface. There are many potential reasons to provide this sort of abstraction. Apple's primary interest is in making it easy and painless for our internal and external developers to improve the ecosystem of Apple products by creating good and secure programs and libraries; subtle deployment problems and/or unnecessary dependencies on the behavior of our implementations would work against these goals.

Almost all languages provide some amount of abstraction of implementation. For example, functions are usually opaque data types which fully abstract away the exact sequence of operations performed. Similarly, adding a new field to a C struct does not break programs which refer to a different field -- those programs may need to be recompiled, but once recompiled, they should continue to work. (This would not necessarily be true if, say, fields were accessed by index rather than by name.)

Components

Programs and libraries are not distributed as a legion of source files assembled independently by the end user. Instead, they are packaged into larger components which are distributed and loaded as a unit. Each component that comprises a program may depend on some number of other components; this graph of dependencies can be assumed to be acyclic.

Because a component is distributed as a unit, ABI resilience within the component is not required. It may still help to serve as a build-time optimization, but Swift aims to offer substantially better build times than C/C++ programs due to other properties of the language (the module system, the lack of a preprocessor, the instantiation model, etc.).

Components may be defined as broadly as an entire operating system or as narrowly as the build products of an individual team. The development process of a very large product (like an operating system) may discourage a model which requires almost the entire system to be recompiled when a low-level component is changed, even if recompilation is relatively fast. Swift aims to reduce the need for unnecessary abstraction penalties where possible, but to allow essentially any model that conforms to the basic rule of acyclicity.

Abstraction Throughout the Program Lifecycle

Most languages provide different amounts of abstraction at different stages in the lifecycle of a program. The stages are:

1. Compilation, when individual source files are translated into a form suitable for later phases. The abstractions of C/C++ object layout and C++ virtual table layout are only provided until this stage; adding a field to a struct or a virtual method to a class potentially require all users of those interfaces to be recompiled.
2. Bundling, when compiled source files are combined into a larger component; this may occur in several stages, e.g. when object files are linked into a shared library which is then bundled up into an OS distribution. Objective-C classes in the non-fragile ABI are laid out by special processing that occurs during bundling; this is done just as an optimization and the runtime can force additional layout changes at execution time, but if that were not true, the abstraction would be lost at this stage.
3. Installation, when bundled components are placed on the system where they will be executed. This is generally a good stage in which to lose abstraction if implementations are always installed before interfaces. This is the first stage whose performance is visible to the end user, but unless installation times are absurdly long, there is little harm to doing extra processing here.
4. Loading, when a component is loaded into memory in preparation for executing it. Objective-C performs class layout at this stage; after loading completes, no further changes to class layout are possible. Java also performs layout and additional validation of class files during program loading. Processing performed in this stage delays the start-up of a program (or plugin or other component).
5. Execution, when a piece of code actually evaluates. Objective-C guarantees method lookup until this "stage", meaning that it is always possible to dynamically add methods to a class. Languages which permit code to be dynamically reloaded must also enforce abstraction during execution, although JIT techniques can mitigate some of this expense. Processing performed in this stage directly slows down the operation.

Expressivity

Certain language capabilities are affected by the choice of when to break down the abstraction.

- If the language runtime offers functions to dynamically explore and, by reflection, use the language structures in a component, some amount of metadata must survive until execution time. For example, invoking a function would require the signature and ABI information about the types it uses to be preserved sufficiently for the invocation code to reassemble it.

- If the language runtime offers functions to dynamically change or augment the language structures in a component, the full abstractions associated with those structures must be preserved until execution time. For example, if an existing "virtual" method can be replaced at runtime, no devirtualization is permitted at compile time and there must be some way to at least map that method to a vtable offset in all derived classes; and if such a method can be dynamically added, there must be some ability for method dispatch to fall back on a dictionary.
- The above is equally true for class extensions in components which may be loaded or unloaded dynamically.

Performance

The performance costs of abstraction mostly take three forms:

- the direct cost of many small added indirections
- code-size inflation due to the extra logic to implement these operations
- diminished potential for more powerful optimizations such as inlining and code motion

As mentioned before, we can avoid these costs within components because classes cannot be extended within components.

We wish to avoid these costs wherever possible by exploiting the deployment properties of programs.

Summary of Design

Our current design in Swift is to provide opt-out load-time abstraction of implementation for all language features. Alone, this would either incur unacceptable cost or force widespread opting-out of abstraction. We intend to mitigate this primarily by designing the language and its implementation to minimize unnecessary and unintended abstraction:

- Within the component that defines a language structure, all the details of its implementation are available.
- When language structures are not exposed outside their defining components, their implementation is not constrained.
- By default, language structures are not exposed outside their defining components. This is independently desirable to reduce accidental API surface area, but happens to also interact well with the performance design.
- Avoiding unnecessary language guarantees and taking advantage of that flexibility to limit load-time costs.

We also intend to provide tools to detect inadvertent changes in interfaces.

Components

(This is just a sketch and deserves its own design document.)

Swift will have an integrated build system. This serves several purposes:

- it creates a "single source of truth" about the project that can be shared between tools,
- it speeds up compiles by limiting redundant computation between files, and
- it gives the compiler information about the boundaries between components.

In complex cases, the build process is going to need to be described. Complex cases include:

- complex component hierarchies (see below)
- the presence of non-Swift source files (to support directly: .s, .c, .o, maybe .m, .mm, .cpp)
- a build product other than an executable (to support directly: executable, .dylib (.framework?), .o, maybe some binary component distribution)
- library requirements
- deployment requirements
- compilation options more complicated than -On

This specification file will basically function as the driver interface to Swift and will probably need a similar host of features, e.g. QA overrides, inheritance of settings from B&I. Some sort of target-based programming may also be required.

Components may be broken down into hierarchies of subcomponents. The component graph must still be acyclic.

Every component has a resilience domain , a component (either itself or an ancestor in its component hierarchy) outside of which resilience is required. By default, this is the top-level component in its hierarchy.

Access

(sketch)

A lot of code is not intended for use outside the component it appears in. Here are four levels of access control, along with their proposed spellings:

- [api] accessible from other components
- [public] accessible only from this component (may need finer grades of control to deal with non-trivial component hierarchies, e.g. public(somecomponent))
- [private] accessible only from this source file
- [local] accessible only from things lexically included in the containing declaration (may not be useful)

A language structure's accessibility is inherited by default from its lexical context.

The global context (i.e. the default accessibility) is [public], i.e. accessible from this component but not outside it.

A language structure which is accessible outside the component it appears in is said to be exported.

Resilience

In general, resilience is the ability to change the implementation of a language structure without requiring further pre-load-time processing of code that uses that structure and whose resilience domain does not include the component defining that structure.

Resilience does not permit changes to the language structure's interface. This is a fairly vague standard (that will be spelled out below), but in general, an interface change is a change which would cause existing code using that structure to not compile or to compile using different formal types.

Language structures may opt out of resilience with an attribute, [fragile]. Deployment versions may be associated with the attribute, like so: [fragile(macosx10.7, ios5)]. It is an interface change to remove an [fragile] attribute, whether versioned or unversioned. It is an interface change to add an unversioned [fragile] attribute. It is not an interface change to add a versioned [fragile] attribute. There is also a [resilient] attribute, exclusive to any form of [fragile], to explicitly describe a declaration as resilient.

Resilience is lexically inherited. It is not lexically constrained; a resilient language structure may have fragile sub-structures and vice-versa. The global context is resilient, although since it is also [public] (and not [api]), objects are not in practice constrained by resilience.

We intend to provide a tool to automatically detect interface changes.

Properties of types

A deployment is an abstract platform name and version.

A type exists on a deployment if:

- it is a builtin type, or
- it is a function type and its argument and result types exist on the deployment, or
- it is a tuple type and all of its component types exist on the deployment, or
- it is a struct, class, or enum type and it does not have an [available] attribute with a later version for a matching platform name.

It is an interface change for an exported type to gain an [available] attribute.

A type is empty if it has a fragile representation (defined below) and:

- it is a tuple type with no non-empty component types, or
- it is a struct type with no non-empty fields, or
- it is an enum type with one alternative which either carries no data or carries data with an empty type.

A type has a fragile representation if:

- it is a builtin type. The representation should be obvious from the type.
- it is a function type. The representation is a pair of two pointers: a valid function pointer, and a potentially null retainable pointer. See the section on calls for more information.
- it is a tuple type with only fragilely-represented component types. The representation of a tuple uses the Swift struct layout algorithm. This is true even if the tuple does not have a fragile representation.
- it is a class type (that is, a reference struct type). The representation is a valid retainable pointer.
- it is a fragile struct type with no resilient fields and no fields whose type is fragilely represented. The representation uses the Swift struct layout algorithm.

A type has a universally fragile representation if there is no deployment of the target platform for which the type exists and is not fragilely represented. It is a theorem that all components agree on whether a type has a universal fragile representation and, if so, what the size, unpadded size, and alignment of that type is.

Swift's struct layout algorithm takes as input a list of fields, and does the following:

1. The fields are ranked:
    - The universally fragile fields rank higher than the others.
    - If two fields A and B are both universally fragile,
    - If no other condition applies, fields that appear earlier in the original sequence have higher rank.
2. The size of the structure is initially 0.

    representations and A's type is more aligned than B's type, or otherwise if A appears before B in the original ordering.
3. Otherwise. Field A is ranked higher than Field B if
    - A has a universal fragile representation and B does not, or

Swift provides the following types:

A language structure may be resilient but still define or have a type

In the following discussion, it will be important to distinguish between types whose values have a known representation and those which may not.

Swift provides

For some structures, it may be important to know that the structure has never been deployed resiliently, so in general it is considered an interface change to change a

Resilience affects pretty much every language feature.

Execution-time abstraction does not come without cost, and we do not wish to incur those costs where unnecessary. Many forms of execution-time abstraction are unnecessary except as a build-time optimization, because in practice the software is deployed in large chunks that can be compiled at the same time. Within such a resilience unit , many execution-time abstractions can be broken down. However, this means that the resilience of a language structure is context-dependent: it may need to be accessed in a resilient manner from one resilience unit, but can be accessed more efficiently from another. A structure which is not accessible outside its resilience unit is an important exception. A structure is said to be exported if it is accessible in some theoretical context outside its resilience unit.

A structure is said to be resilient if accesses to it rely only on its

A structure is said to be universally non-resilient if it is non-resilient in all contexts in which it is accessible.

Many APIs are willing to selectively "lock down" some of their component structures, generally because either the form of the structure is inherent (like a point being a pair of doubles) or important enough to performance to be worth committing to (such as the accessors of a basic data structure). This requires an [unchanging] annotation and is equivalent to saying that the structure is universally non-resilient.

Most language structures in Swift have resilience implications. This document will need to be updated as language structures evolve and are enhanced.

Type categories

For the purposes of this document, there are five categories of type in Swift.

Primitive types: i1, float32, etc. Nominal types defined by the implementation.

Functions: () -> int, (NSRect, bool) -> (int, int), etc. Structural types with

Tuples: (NSRect, bool), (int, int), etc. Structural product types.

Named value types: int, NSRect, etc. Nominal types created by a variety of language structures.

Named reference types: MyBinaryTree, NSWindow, etc. Nominal types created by a variety of language structures.

Primitive types are universally non-resilient.

Function types are universally non-resilient (but see the section on calls).

Tuple types are non-resilient if and only if all their component types are non-resilient.

Named types declared within a function are universally non-resilient.

Named types with the [unchanging] annotation are universally non-resilient. Problem, because of the need/desire to make things depend on whether a type is universally non-resilient. Makes it impossible to add [unchanging] without breaking ABI. See the call section.

All other named types are non-resilient only in contexts that are in the same resilient unit as their declaring file.

Storage

Primitive types always have the primitive's size and alignment.

Named reference types always have the size and alignment of a single pointer.

Function types always have the size and alignment of two pointers, the first being a maximally-nonresilient function pointer (see the section on calls) and the second being a retain/released pointer.

If a tuple type is not universally non-resilient, its elements are stored sequentially using C struct layout rules. Layout must be computed at runtime. Separate storage is not really a feasible alternative.

Named types

It is an error to place the [unchanging] annotation on any of these types:

- a struct type with member types that are not universally non-resilient
- an enum type with an enumerator whose type is not universally non-resilient
- a class extension
- a class whose primary definition contains a field which is not universally non-resilient

Classes

It is an error to place the [unchanging] annotation on a class extension.

It is an error to place the [unchanging] annotation on a class whose primary definition contains a field whose type is potentially resilient in a context where the class is accessible. That is, if the class is exported, all of its fields must be universally non-resilient. If it

is not exported, all of its fields must be non-resilient within its resilience unit.

It is allowed to add fields to an [unchanging] class in a class extension. Such fields are always side-stored, even if they are declared within the same resilience unit.

Objects

Right now, all types in Swift are "first-class", meaning that there is a broad range of generic operations can be

1. the size and layout of first-class objects:
    - local variables
    - global variables
    - dynamically*allocated objects
    - member sub*objects of a structure
    - base sub*objects of a class
    - element sub*objects of an array
    - parameters of functions
    - results of functions
2. the set of operations on an object:
    - across all protocols
    - for a particular protocol (?)
3. the set of operations on an object
    - ...

ABI resilience means not making assumptions about language entities which limit the ability of the maintainers of those entities to change them later. Language entities are functions and objects. ABI resilience is a high priority of Swift.

- functions
- objects and their types

We have to ask about all the

Notes from meeting.

We definitely want to support resilient value types. Use cases: strings, dates, opaque numbers, etc. Want to lock down API even without a concrete implementation yet.

This implies that we have to support runtime data layout. Need examples of that.

We do need to be resilient against adding [unchanging]. Okay to have two levels of this: [born_unchanging] for things that are universally non-resilient, [unchanging] for things that were once resilient. Proposed names: [born_fragile] and [fragile].

Global functions always export a maximally-resilient entrypoint. If there exist any [fragile] arguments, and there do not exist any resilient arguments, they also export a [fragile] copy. Callers do... something? Have to know what they're deploying against, I guess.

Want some concrete representation for [ref] arguments.

Notes from whiteboard conversation with Doug.

What does fragility mean for different kinds of objects?

structs (value types) - layout is fixed

their fields - can access field directly rather than by getter/setter

their methods - like any function

classes (reference types) - layout of this struct component is fixed

their fields - access directly vs. getter/setter

their methods - like any function

class extensions - like classes. what to do about layout of multiple class extensions? very marginal

functions - inlinable

global variables - can be directly accessed. Type is born_fragile: value is global address. Type is resilient: lvalue is load of global pointer. Type is fragile: value is load of global pointer, also optional global address using same mechanism as global functions with fragile argument types

protocols - born_fragile => laid out as vtable. Can these be resilient?

their implementations: contents of vtable are knowable

enums - layout, set of variants

Notes from second meeting

Resilience attributes:

- born_fragile, fragile, resilient
- want to call born_fragile => fragile, fragile => fragile(macosx10.42)
- good except "default", more minimal thing is the more aggressive thing. More important to have an ABI-checking tool
- use availability attributes scheme: platformX.Y.Z

Components: Very much a build-system thing.

Users should be able to write swift [foo.swift]+ and have it build an executable.

For anything more complicated, need a component description file.

- hierarchy of components
- type of build product: executable, dylib, even object file
- non-Swift sources (object files, C files, whatever)
- deployment options (deploying on macosxX.Y.Z)
- need some sort of "include this subcomponent" declaration
- probably want some level of metaprogramming, maybe the preprocessor?

Host of ways of find the component description file automatically: name a directory (and find with fixed name), name nothing (and find in current directory)

Component organization is the seat of the decision algorithm for whether we can access something resilient fragilely or not.

- not necessarily just "are you in my component"; maybe "are you in my domain/component tree/whatever"

Resilience is lexically inherited.

- Declarations inside a fragile enum are implicitly fragile, etc.
- Except anything inside a function is fragile.

Break it down by types of declarations.

- typealias has no resilience
- struct -- the set/order of fields can change -- means size/alignment, layout, copy/destruction semantics, etc. can all change
- fields - direct access vs. getter/setter
- funcs - as if top level
- types - as if top level
- class -- same as a structs, plus
- base classes -- can't completely remove a base class (breaks interface), but can introduce a new intermediate base
- virtual dispatch -- table vs. dictionary, devirtualization (to which decl?). Some amount of table lookup can be done as static vs. dynamic offsets
- funcs -- inlineability
- vars -- direct access vs. getter/setter. Direct accesses for types that aren't inherently fragile need to be indirected because they may need to be dynamically allocated. In general, might be actor-local, this is for when the model does say "global variable".
- extensions of classes -- like class. Fields are always side-allocated if we're extending a class not defined in this component (w/i domain?). Making a class fragile is also a promise not to add more fields in extensions in this component; probably need a way to force a side-table.
- protocols -- can't remove/change existing methods, but can add defaulted methods. Doing this resiliently requires load-time checking. vtable for non-defaulted methods, ? for rest?
- enum - set of directly represented cases
- enum elements - directly represented vs. injection/projection.
- enum - called out so that we can have an extensible thing that promises no data fields. Always an i32 when resilient.
- const - fragile by default, as if a var otherwise