Interoperability between Swift and C++

This document discusses the design and tradeoffs for bidirectional API-level interoperability between Swift and C++.

Assumptions:

- We can make changes to the Swift language and the standard library.
  - The proposed changes must fit Swift's goals and philosophy. In other words, the proposed changes must have a reasonable chance to be accepted by the Swift community. For example, a change that requires an ABI break on Apple platforms is a non-starter.
  - Forking the Swift language or standard library, or creating a dialect without a fork (and therefore, being able to make radical changes to Swift's goals, philosophy, safety, or ergonomics) are not interesting options, and therefore, are not considered or discussed.
- We can make limited changes to the C++ code, toolchain, standard library implementation, and runtime environment.
  - The cost of such changes must be taken into account. Changes that require an ABI break for C++ might be interesting for users who control the complete toolchain, but are non-starters for the community at large; therefore, such changes can only be considered an optimization. Changes that require significant manual work on existing C++ code by end users can also be considered only as optimizations or improvements for ergonomics.
- Goals
- Current state of art: importing C code into Swift
- Importing C++ APIs into Swift
  - Names, identifiers and keywords
  - Functions
    - Non-const pointer and reference parameters
    - Const reference parameters
    - Mapping overload sets
    - Inline functions
  - Namespaces and modules
    - The "std" namespace
  - Structs and classes
    - Mapping the type itself and its special member functions
    - Special member functions with incorrect semantics
    - Special member functions that throw exceptions
    - Precise destruction semantics
    - Member functions that return inner pointers
    - Differences in move semantics between C++ and Swift
    - Move-only C++ classes
    - Move-only C++ classes: mapping in Swift 5
    - Move-only C++ classes: mapping to move-only Swift types (in future Swift versions)

        * Non-destructive moves
        * Non-movable C++ classes
        * Member functions
        * Const-qualified member functions in C++ classes
        * Volatile-qualified member functions
        * Ref-qualified member functions
        * Final functions
        * Getters and setters
        * Pointers to members
        * Virtual member functions
        * Inherited APIs
        * Conversions between derived and base classes
        * Subclassing C++ classes in Swift
        * Making C++ classes conform to Swift protocols
    - Enums
    - Templates
        * Function templates
        * Function templates: import as Swift generic functions
        * Function templates: allow to specify template arguments
        * Function templates: calls to specific specializations
        * Function templates: calls with generic type parameters
        * Function templates: importing as real generic functions
        * Class templates
        * Class templates: importing instantiation behind typedef
        * Class templates: importing specific specializations
        * Class templates: using with generic type parameters
        * Class templates: using in generic code through a synthesized protocol
        * Class templates: importing as real generic structs
        * Mapping SFINAE
        * Variadic templates
        * Non-type template parameters
        * Template template parameters
    - Overloaded operators
    - `operator*`, `operator->`
    - `operator[]`
    - `operator()`
    - Literal operators
    - Exceptions
    - Atomics
    - Importing non-const pointer as extra return values
- Enhancing C++ API mapping into Swift with bridging
    - Bridging data types with different memory layout
    - Bridging: behind the scenes
    - Bridging does not work in every context
    - Bridging `std::function`

  - Bridging `std::string`
  - Bridging `std::vector`
  - Bridging `std::set`, `std::map`
  - Bridging `std::unordered_set`, `std::unordered_map`
  - Bridging `std::tuple`, `std::pair`
  - Bridging `std::span`
  - Standard C++ containers in general
  - Custom C++ containers
  - Bridging `absl::flat_hash_{set,map}`, `absl::node_hash_{set,map}`
- Enhancing C++ API mapping into Swift with annotations
  - Annotation workflow
  - Tooling to infer nullability annotations
  - APIs that take a pointer and count
- Current state of art: importing Swift code into C
- Importing Swift APIs into C++
  - Resilient types
- Forum discussions

Goals

What are the properties of a good interoperability layer? Exercising interoperability is not in itself a goal of any Swift or C++ user. Therefore, for API users interoperability should be maximally transparent in all aspects: API design and ergonomics, editor integration and tooling, and performance. For API vendors, interoperability should not be a significant extra burden, while allowing API vendors to own and curate the imported API surface. Let's discuss what these points mean.

API design and ergonomics. Ideally, users working in Swift should not feel any difference between native Swift APIs and imported C++ APIs.

For example, while it is possible to write a custom hashtable in Swift, it is done extremely rarely, and most code uses the vocabulary types `Set` and `Dictionary`. Therefore, if C++ APIs that used `std::unordered_map` or `flat_hash_map` types, when imported in Swift, would continue using those C++ map types, they would look weird and foreign in Swift. Idiomatic Swift code that uses `Set` and `Dictionary` will have to convert the data into these foreign hashtable types before calling imported C++ APIs.

As another example, C++ functions often receive values by const reference or by const pointer. Semantically, the closest equivalent in Swift is `UnsafePointer`. However, `UnsafePointer` is not as common in Swift as const references are in C++; it is not idiomatic for Swift APIs to accept `UnsafePointer` outside of a couple exceptional situations (for example, implementations of low-level facilities, performance-sensitive code, etc.) Therefore, if C++ APIs, when imported in Swift, would use `UnsafePointer` a lot, they would look weird and foreign in

Swift. Imported C APIs, when they are not operating on Objective-C types, already use `UnsafePointer` a lot, and look non-idiomatic because of that. Swift provides some affordances that make calling them easier.

Editor integration and tooling should transparently support code in both languages. To the extent possible, all editor operations should "see through" the interop. For example, if the user invokes the "rename function" refactoring, it should rename all definitions, declarations, and usages across Swift and C++ code. This goal has been largely achieved for Swift/Objective-C interop, and we are planning to rely on the same mechanisms for Swift/C++ interop.

Performance. C++ APIs, when used from Swift, ideally, should have performance characteristics identical to when they are used from C++.

Interop must not be a burden for API vendors. Enabling Swift code to call a certain C++ library should create minimal burden for the owners of that C++ library. Ideally, it should be possible without any involvement of the owners.

For example, requiring the API owners to create an API description file (like in CLIF for Python), or a glue layer (like in JNI for Java) is a significant burden. Most API owners are not going to do it without specific requests from users, and even when such request is received, many API owners will carefully consider if they want to take on maintenance of this extra file.

It may be possible to allow users to do the work necessary to expose a C++ library to Swift, however, that is not a great option either. A C++ library might end up with multiple (incompatible) bindings of varying quality, covering different parts of the API. In addition, the owner of the API loses control of the API surface.

Allow API vendors to own and curate the imported API surface. While the minimal amount of work to expose a C++ library to Swift should be ideally "none" (it should just work), API vendors should own the Swift API surface of their C++ libraries, and should be able to adjust it where the automated interop is not satisfactory.

Tension and conflicts between goals. These goals are conflicting. For example, API ergonomics are in conflict with performance: we can provide a more ergonomic API if we automatically bridge C++ types to corresponding Swift vocabulary types at the API boundary, however, these type conversions will cost CPU cycles. The solutions will be designed on a case by case basis: sometimes, we will pick one side of the tradeoff, sometimes we will pick one side but allow the user to override the default, and sometimes we will add multiple facilities that pick different sides, forcing the user to choose.

Current state of art: importing C code into Swift

Swift/C++ interoperability builds on top of the Swift/C interoperability, so it helps to be familiar with Swift's strategy for importing C modules.

Importing C++ APIs into Swift

Now let's talk about extending existing rules to handle C++.

Names, identifiers and keywords

C++ allows defining names that are not identifiers:

- Destructors
- Overloaded operators
- Literal operators

Member functions with non-identifier names are discussed in the section about classes. Other constructs are discussed in their respective sections.

Functions

Non-const pointer and reference parameters

In C++, there are no language requirements for pointers passed as function arguments. For example, given a function signature like this:

```
void increment(int *value);
```

C++ as a language poses no requirements for `value` to live long enough during the function call, or even that `value` points to valid memory when the function is called; `value` can be null or it can be an invalid (for example, freed) pointer. There are also no requirements about the memory pointed to by `value` being initialized, or valid to dereference without data races. Pointers are also allowed to alias other memory, as long as the types are compatible. Pointers also don't imply anything about ownership or non-ownership.

Rules in C++ about references are a bit tighter, but not much. The only differences are that a reference can't be null, and that it must be bound to a valid object when it is initially created.

In practice, engineers working in C++ often make the following assumptions about third-party APIs, even though these properties are not guaranteed by the language, and are often not explicitly documented by the API:

- All pointers passed to a function are valid to dereference.
- Almost all pointers passed to a function remain valid to dereference while the function is running.

- Almost all pointers passed to a function are valid to dereference without data races.
- Almost all pointers passed to a function point to initialized data. Passing a pointer to uninitialized memory and expecting the function to initialize it happens occasionally, but it is not common.
- Often enough, pointers passed to a function are not null.
- Often enough, pointers of different types don't alias each other's subobjects.
- Almost all `const &T` parameters are semantically equivalent to passing `T` by value.

In Swift, passing unsafe pointers as function arguments is not idiomatic. Idiomatic code passes structs by value, and classes by reference. `inout` parameters in Swift allow functions to read and modify storage specified by the caller. The argument for the `inout` parameter must be a storage reference expression.

```
func increment(_ value: inout Int) {
  value += 1
}

struct TwoInts {
  var x: Int = 0
  var y: Int = 0
}

func caller() {
  var ints = TwoInts()
  increment(&ints.x)
  // ints.x is 1
  // ints.y is 0
}
```

To understand the constraints that Swift puts on `inout` parameters, let's take a look at the mental model for introduced in the Ownership manifesto and in SE-0176 Enforce Exclusive Access to Memory. When the caller binds a storage reference to an `inout` parameter, it starts a non-instantaneous access to the whole value that occupies the storage. This access ends when the callee returns. Overlapping accesses are not allowed, and therefore, while an `inout` parameter is live, original value may not be accessed.

(Note that before the ownership manifesto, `inout` parameters were explained in terms of a different model that might be easier to understand. Nevertheless, both models are equivalent. The old model stated that on function entry, the value is moved from the caller-specified storage reference into the `inout` parameter, and moved from the `inout` parameter back into the original storage on function exit. During the execution of the function, the original storage remains uninitialized,

and therefore, the whole value that occupied that storage remains inaccessible until the function returns.)

In practice, `inout` parameters are implemented as passing a pointer to the storage that should be mutated, while maintaining behavior identical to the models explained above. As a consequence of these models, `inout` parameters provide the following guarantees:

- the memory backing an `inout` parameter is valid to access throughout the execution of a function.
- since `inout` parameters are not pointers, there is no such thing as "null `inout` parameter".
- `inout` parameters are valid to read and modify throughout the execution of a function without data races, unless the function itself has shared the parameter with multiple threads.
- `inout` are backed by initialized memory on function entry and function exit (an implication of the copy-in/copy-out semantics). Destroying the object in `inout` requires using unsafe constructs. Therefore, in safe Swift `inout` parameters are backed by initialized memory throughout function execution.
- `inout` parameters don't alias each other or any other values that program is allowed to access at that point. Remember that the original value that is bound to an `inout` parameter is inaccessible for the duration of the `inout` binding.

In practice, non-const pointers and references in function parameters are most often used for the same purpose as Swift's `inout`, so it is desirable to map them to each other.

```
// C++ header.

void incrementBoth(int *value1, int *value2);

// C++ header imported in Swift (possible mapping).

func incrementBoth(_ value1: inout Int, _ value2: inout Int)
```

The `incrementBoth` function imported like this has more restrictions in Swift than in C++ (for example, arguments may not alias). Calling this C++ function from Swift does not create a new kind of safety issues in Swift, because a language with more restrictions calls a language with fewer restrictions. From the API point of view, it should not be a problem either, unless the caller needs to pass arguments that alias (Swift's enforcement of exclusivity will prevent that).

Based on the comparison above, it looks like Swift's `inout` provides strictly more guarantees than C++'s non-const pointers and references. Therefore, it is not

safe to map them to each other in all cases. For example, we can't apply this mapping when Swift is implementing a function exposed to C++, for example:

```
// C++ header.

class Incrementer {
public:
  virtual void incrementBoth(int *value1, int *value2) = 0;
};

// C++ header imported in Swift (possible mapping).

protocol Incrementer {
  func incrementBoth(_ value1: inout Int, _ value2: inout Int)
}

struct MyIncrementer: Incrementer {
  func incrementBoth(_ value1: inout Int, _ value2: inout Int) {
    // The language requires that `value1` and `value2` don't alias.
  }
}
```

It would be unsafe to expose an Swift implementation of `Incrementer.incrementBoth` through the C++ signature, unless the C++ function already has Swift-like preconditions for pointer arguments. For true inout parameters in C++, those preconditions should in fact hold, even if the C++ function does not formally document them.

Const reference parameters

In C++, const references in function arguments are most often used to avoid copying objects passed to functions. Swift solves this problem in two ways:

- by providing language features that allow the engineer to introduce indirections (for example, reference types, existential types, indirect enum cases, tools for defining copy-on-write types),
- by automatically passing large arguments indirectly without copying them, where possible.

It is not feasible to automatically map C++ const references to Swift-specific indirections like class types, because that would require changing the memory layout through non-trivial bridging.

Option 1: Import `const T&` as `UnsafePointer<T>`.

We can easily map C++ const references in function parameters to `UnsafePointer<T>` in Swift, however, the resulting APIs will not look idiomatic. They will be usable from Swift, because Swift code can make an unsafe pointer with the `&` operator, however, it can only be applied to mutable storage locations.

Therefore, more variables will be mutable in Swift than necessary. Furthermore, to make some values mutable, the code will need to make a copy that will be stored in a mutable variable – defeating the point of the C++ API taking a const reference.

```
void printInt(const int &value);

// C++ header imported in Swift.

func printInt(_ value: UnsafePointer<Int>)

// Usage example.

func caller() {
  var x = 42
  printInt(&x) // OK

  let y = 42
  printInt(y) // error: type mismatch: `Int` is not an `UnsafePointer<Int>`
  printInt(&y) // error: can't apply `&` to an immutable value
}
```

Improvement for option 1: Allow using `&` on immutable values.

To eliminate the extra copies that the code would have to perform to make some values mutable, we could extend the Swift language to allow applying `&` to immutable values, producing an `UnsafePointer<T>`.

```
// Usage example.

void caller() {
  let x = 42
  printInt(&x) // OK
}
```

Option 2: Import `const T&` as `T`.

A more ergonomic approach would be to import C++ const references to value types as values, but still use a by-reference calling convention behind the scenes.

```
void printInt(const int &value);

// C++ header imported in Swift.

func printInt(_ value: Int)

// Usage example.

func caller() {
  let x = 42
```

```
  printInt(y) // OK
}
```

The imported `printInt` function that takes an `Int` by value would be a thunk that calls the actual C++ entry point that takes an `Int` by a const reference. A mandatory optimization pass would inline that thunk into its callers, and eliminate the copies that now became unnecessary.

It is important to understand that this approach only works in cases where the compiler can insert a thunk. Here's an example where the compiler can't transparently wrap a function pointer into a thunk, and therefore can't import `const int&` as `Int32`:

```
struct Example {
  void (*printInt)(const int &);
};
```

This approach is much more ergonomic. However, it has a few disadvantages.

One disadvantage is that this approach provides less control: the caller can't specify or predict the exact address that is passed to the C++ function (because Swift can make copies of values or move them around in memory). Most functions receiving `const T&` parameters should not care about the exact address, but some might.

Another disadvantage of this approach is that it is safe in many cases in practice, but users can construct unsafe cases. Specifically, if the C++ code being called persists the reference that was passed by Swift beyond the duration of the function call, that reference can become invalid (because Swift code could pass an address of a temporary, implicitly materialized by the compiler). The nature of unsafety is identical between option 1, option 2, and existing Swift/C interop, however, in option 2, where we import C++ references as Swift values, it is not visible to the Swift programmer that the function being called accepts an address.

Mapping overload sets

C++ has complex overload resolution rules, in part to support certain API design patterns. Some API design patterns appeared as a consequence of overload resolution rules. Therefore, in C++ an "API atom" is an an overload set, not an individual function (CppCon 2018: Titus Winters "Modern C++ Design (part 1 of 2)"). The same is also the case in Swift, so, fundamentally, there is no impedance mismatch here.

However, C and Objective-C interoperability imports each function and method separately. For C++ interoperability, we should recognize design patterns that use overload sets, and map them appropriately to Swift.

One such C++ pattern is providing two overloads: one that takes a `const T&` and another that takes a `T&&`. If we attempt to naively map them to Swift, they

would map to the same signature. If we import just one of them and ignore the other, we would be leaving performance on the table.

The easiest way to map such overload sets is to import the `T&&` overload as a function that takes the argument by value, and ignore the `const T&` overload. This approach does create a small amount of performance overhead (an extra move), but does not require changing the Swift language or type checker to recognize the concept of temporaries. The SIL optimizer seems like the appropriate place to opportunistically reclaim that performance, by removing unnecessary moves of temporaries where possible, and replacing calls to the thunk with calls to the most appropriate C++ entry point.

```
// C++ header.

struct Tensor { ... };

void processTensor(const Tensor&);
void processTensor(Tensor&&);
```

```
// C++ header imported in Swift.

struct Tensor { ... }

func processTensor(_: Tensor) {
  // call `void processTensor(Tensor&&)`
}
```

```
// Usage example.

func useTensor() {
  var x = Tensor()
  processTensor(x)
  processTensor(x)
}
```

```
// Equivalent of SIL code after optimizations.

// void processTensor(const Tensor&);
func processTensorByConstRef(_: UnsafePointer<Tensor>)

// void processTensor(Tensor&&);
func processTensorByRvalueRef(_: UnsafeMutablePointer<Tensor>)

func useTensor() {
  var x = Tensor()
  processTensorByConstRef(x)
  processTensorByRvalueRef(x) // Automatically move the value because it is obviously not us
}
```

Once move-only types are added to Swift, the same mapping technique becomes applicable to C++ move-only value types as well, with the only difference that argument of the imported function would be marked as being consumed.

Inline functions

Inline functions (both free and member functions) are used in C++ a lot more than inline functions in C. For inline functions in C++, Swift should use the same strategy as it uses currently for C: use Clang's CodeGen libraries to emit definitions of inline functions into one LLVM module together with Swift code.

Namespaces and modules

Namespaces and modules in C++ are orthogonal concepts. Namespaces in C++ can span multiple modules; C++ namespaces can also be nested. In Swift, modules are namespaces.

Option 1: Map C++ namespaces to empty enums in Swift.

There are precedents for using empty enums as namespaces in Swift (for example, `CommandLine`, `Unicode`, `Unicode.UTF8` in the standard library). However, empty enums are not a perfect substitute for C++ namespaces:

- Swift has no `using`-like construct that allows code to avoid qualifying names nested in an enum (in other words, names nested within an enum must be always qualified with the name of that enum when used).
- C++ namespaces can span multiple modules. We don't want every imported C++ module to define its own empty enum, as that will lead to name collisions, which will require even more qualification from users.

```
// C++ header in module `CppButton`.

namespace widgets {
class Button {};
}

// C++ header in module `CppTextbox`.

namespace widgets {
class Textbox {};
}

// C++ module `CppButton` imported to Swift.

enum widgets {
  struct Button {}
}
```

```
// C++ module `CppTextbox` imported to Swift.

enum widgets {
  struct Textbox {}
}

// Usage example: everything works well when we import only `CppButton`.

import CppButton

func makeButton() {
  var b1 = Button() // error: no such type
  var b2 = widgets.Button() // OK
}

// Usage example: ambiguities when we import both `CppButton` and `CppTextbox`.

import CppButton
import CppTextbox

func makeButton() {
  var b1 = Button() // error: no such type
  var b2 = widgets.Button() // error: name `widgets` is ambiguous, did you mean `CppButton.
  var b3 = CppButton.widgets.Button()
}
```

Improvement for option 1: Move all empty enums to one synthetic module.

We could fix the ambiguity between same-named namespace enums defined by multiple C++ modules by synthesizing an extra Swift module that will contain only the enums that make up the namespace structure, and then making all other C++ modules define extensions for those enums.

```
// Swift module `CppNamespaces` synthesized by the C++ importer.

enum widgets {}

// C++ module `CppButton` imported to Swift.

import CppNamespaces

extension widgets {
  struct Button {}
}

// C++ module `CppTextbox` imported to Swift.

import CppNamespaces
```

```
extension widgets {
  struct Textbox {}
}

// Usage example.

import CppButton
import CppTextbox
// Implicitly injected: import CppNamespaces

func makeButton() {
  var b1 = Button() // error: no such type
  var b2 = widgets.Button() // OK
}
```

An incidental readability advantage of using enum extensions is that the pretty-printed module interface for C++ modules will use the keyword `extension` for namespaces instead of `enum`, reducing the confusion potential.

Improvement for option 1: Add a `using`-like construct to Swift.

Some C++ libraries define namespaces with long names, or deeply nested namespaces. C++ users of such libraries often prefer to avoid typing and seeing such namespace qualifiers in their code. To help use these libraries in Swift, we could add a construct similar to C++ `using`, that would expand the name lookup into the given empty enum.

```
// Usage example.

import CppButton
import CppTextbox
// Implicitly injected: import CppNamespaces
import CppNamespaces.widgets

func makeButton() {
  var b1 = Button() // OK
}
```

We could limit this construct to only work with enums in the `CppNamespaces` module, if such feature is not deemed to be appropriate for APIs defined in Swift.

Option 2: Introduce namespaces into Swift as a C++ interoperability feature.

From the implementation point of view, option 1 feels like a pile of hacks and tricks. We could add C++-style namespaces to Swift, and only allow the C++ importer to define them, clearly designating namespaces as a C++

interoperability feature. From the point of view of users, there will be very few, if any, differences between option 1 and option 2.

Option 3: Introduce namespaces into Swift.

We could add C++-like namespaces to Swift as a generally available language feature, and then use it to import C++ into Swift.

The "std" namespace

An interesting question is how to import the `std` namespace in Swift. Continuing to call it `std` might invoke incorrect associations, because APIs in `std` are not a part of the Swift standard library. We could rename `std` to something that suggests that it is the C++ standard library, like `cxx_std`.

Structs and classes

Mapping the type itself and its special member functions

For the purposes of this discussion, we will consider C++ classes only. There are no important technical differences between C++ structs and classes, both have exactly the same capabilities. In C++, structs and classes differ in how they are typically used, however, it is not enforced by the compiler. Different style guides have different rules about when to define a struct or a class, and these rules are often not toolable in principle. Therefore, struct/class distinction in C++ can be used at best as a hint to guide the API importer.

Swift's structs and classes are two completely different concepts. Structs are the building block for value types, and classes are the building block for reference types. References to Swift class instances are similar to `std::shared_ptr<T>` in C++ (but are more efficient, because they use an intrusive refcount), and therefore are not suitable for mapping an arbitrary C++ class type.

As far as value/reference semantics go, C++ classes are similar to structs in Swift: both inherit value/reference semantics from constituent parts (if you compose value types in a C++ class or a Swift struct, you get a value type; if you embed a reference type in a C++ class or in a Swift struct, the composite type behaves like a reference type). They are also similar in their memory allocation strategy: both are allocated inline (local variables are allocated on the stack, member variables are allocated in the containing object without an indirection).

C++ classes allow the API owner to separately control whether the value is copyable, movable, copy-assignable, move-assignable, and destructible. Moreover, behavior for those operations can be customized. User-defined structs in Swift can't customize such operations. User-defined Swift classes can only customize the `deinit`, which corresponds to the destructor in C++. Nevertheless, this limitation only applies to user-defined Swift structs and classes. There are more mechanisms in the Swift object model, compiler, and runtime, that are not exposed as language features, but exist as hooks for future extension (e.g., some

were added specifically for potential C++ interop), or exist to support Swift language features like resilience.

Let's take a look at what types look like in the compiler and runtime, and what sorts of customization hooks are available.

The compiler classifies types as loadable or address-only (either one or the other). Address-only types are always manipulated through a pointer. Loadable types can be loaded from memory into SIL and LLVM IR values, and then reconstructed back in memory.

The compiler uses different strategies to access and manage values depending on the classification of the type. For example, trivial loadable types can be freely copied, destroyed, and moved around in memory without running any user-defined code. Structs imported from C are considered loadable in Swift. C structs are only composed of C primitives, and therefore are considered trivial – can be initialized, assigned, and moved around with `memcpy`.

Resilient Swift types are on the complete opposite end of the spectrum. Resilient types are types provided by a third-party library that does not want to commit to a particular ABI for that type. Resilient types are opaque values that must be manipulated through a value witness table – a table of function pointers that defines the following operations:

Swift Value Witness Operation	C++ equivalent
`initializeWithCopy`	copy constructor
`assignWithCopy`	copy assignment operator
`initializeWithTake`	move constructor, followed by a call to destructor on the source
`assignWithTake`	move assignment operator, followed by a call to destructor on the source
`destroy`	destructor
`size`	`sizeof(T)` minus trailing padding
`stride`	`sizeof(T)`
`flags`	among other information, contains alignment, i.e., `alignof(T)`

Value witness tables also contain composite operations that can be derived from the operations above.

We can map an arbitrary copyable C++ class type to a Swift struct type whose value witnesses call the corresponding special member function of the C++ class.

We can map certain C++ classes to a more optimizable representation in Swift. For example, C++ classes that contain only primitives and have trivial special member functions (i.e., POD types) can be mapped to trivial loadable Swift structs, just like C structs are today.

Can we map more complex C++ classes to loadable Swift structs? Someone with a more thorough knowledge of runtime and SIL should answer, but here's what the author of the document could find.

From `swift/docs/SIL.rst`:

> Loadable types are types with a fully exposed concrete representation:
>
> - Reference types
> - Builtin value types
> - Fragile struct types in which all element types are loadable
> - Tuple types in which all element types are loadable
> - Class protocol types
> - Archetypes constrained by a class protocol

"Fully exposed concrete representation" looks like a definition of a loadable type, however, other comments say slightly different things, and it is not clear what "fully exposed" actually means.

From `swift/include/swift/SIL/TypeLowering.h`:

```
/// Is a lowered SIL type address-only?  That is, is the current context
/// required to keep the value in memory for some reason?
///
/// A type might be address-only because:
///
///   - it is not fixed-size (e.g. because it is a resilient type) and
///     therefore cannot be loaded into a statically-boundable set of
///     registers; or
///
///   - it is semantically bound to memory, either because its storage
///     address is used by the language runtime to implement its semantics
///     (as with a weak reference) or because its representation is somehow
///     address-dependent (as with something like a relative pointer).
///
/// An address-only type can be fixed-layout and/or trivial.
/// A non-fixed-layout type is always address-only.
enum IsAddressOnly_t : bool {
  IsNotAddressOnly = false,
  IsAddressOnly = true
};
```

C++ classes are fixed-size, so we don't care about the first bullet point. The second bullet point is more interesting – a type must be address-only if an instance of that type cares about which address it is stored at.

From Slava Pestov's blog post "How to talk to your kids about SIL type use":

> If you define a C++ class describing a smart pointer to a reference-counted object, the presence of a custom copy constructor and de-

> structor force the value to be materialized in memory all of the time, because the compiler doesn't know what the hell you're doing inside the custom constructor; maybe you're taking the address of 'this' and storing it somewhere.
>
> If C++ had loadable types, they would involve some kind of special annotation that told the compiler that the move constructor can just trivially do a memcpy() of the value, even if there is a custom copy constructor or destructor. At this point, your smart pointer could be loaded into a register and passed around without issues.

This example says that a C++ smart pointer, with a custom copy constructor, copy assignment operator, and destructor, can be loadable, as long as all move special member functions are trivial. This explanation aligns well with the previous one.

A C++ class is loadable in Swift as long as moving it is trivial. If we ignore C++ classes with inconsistent copy and move semantics, a trivial move implies that copy constructor and copy assignment don't care about the specific address, which matches Swift's requirements.

Special member functions with incorrect semantics

C++ classes can define special member functions with incorrect or inconsistent semantics. It can be done intentionally; for example, the copy constructor of `auto_ptr` actually moves the value. It can also be done unintentionally: it is common to define only the special member functions that the C++ code happens to use right now – for example, only define the copy constructor but not the copy assignment.

Therefore, we need to decide whether the Swift/C++ interoperability layer should trust special member functions of C++ classes to have expected semantics, or whether some verification is necessary.

Non-user-provided special member functions have correct semantics by construction if the C++ class is composed of well-behaved types. Given the current state of the art in static analysis, it does not seem like the interoperability layer even remotely has a chance of proving that user-provided special member functions have correct semantics. There are two practical options: either trust the user in this regard and allow them to adjust the inference done by the interop layer with annotations, or don't make any assumptions about the semantics and demand that the user annotates types as value types, move-only, or non-movable.

Special member functions that throw exceptions

If C++ classes are imported in Swift as structs, there seems to be no reasonable way to map exceptions thrown from special member functions to an error that can be handled in Swift.

While it is possible to propagate C++ exceptions thrown by normal member functions to Swift code, special member functions are different as they used used to implement value witnesses, which are called by the compiler implicitly. Therefore, either such exceptions have to be mapped to fatal errors (as we do for other unhandled C++ exceptions), or calls to such special member functions must be prevented statically.

Preventing such calls statically seems difficult. Also, in practice, special member functions throw mostly due to out-of-memory conditions, which is considered unrecoverable in Swift. Therefore, the practical solution is to map exceptions from special member functions to fatal errors.

If the user must use a C++ class with throwing special member functions from Swift, a reasonable workaround would be to write a wrapper that exposes these operations as regular APIs. Wrappers can even be synthesized by the importer, if such classes are found to be common.

```
// Can't be imported in Swift.
class MyThrowingType {
public:
  MyThrowingType() { if(...) throw ...; }
  ~MyThrowingType() { if(...) throw ...; }
  void method();
};

// Can be imported in Swift.
class MyThrowingTypeWrapper {
private:
  std::unique_ptr<MyThrowingType> value_;
public:
  MyThrowingTypeWrapper(): value_(nullptr) {}
  void init() /* can throw */ { value_ = std::make_unique<MyThrowingType>(); }
  void deinit() /* can throw */ { value_ = nullptr; }
  void method() { value_.method(); }
};
```

Precise destruction semantics

C++ language specifies the exact point in program execution at which destructors should run, while Swift does not. Some C++ APIs rely on these guarantees to work properly (for example, RAII types like mutex locks).

```
// A pure C++ program.

class PickyCxxClass {
public:
  ~PickyCxxClass() { ... }
};
```

```
void test1() {
  PickyClass c;
  printf("Hello");
  // `c`'s destructor is guaranteed to be called exactly here.
}

// A pure Swift program, similar to the C++ program above.

struct SwiftStruct { ... }

func test1() {
  var s = SwiftStruct()
  // `s` can be deallocated here.
  print("Hello")
  // `s` can also be deallocated here.
}
```

But how commonly do C++ APIs rely on precise destruction? A gut feeling is that it happens pretty often, but it is not always intentional and acknowledged by the API vendor. *All* C++ types get a precise destruction point, and therefore, it is very easy to start inadvertently relying on it in API design.

There are multiple ways to deal with this impedance mismatch.

Ignore the differences, and apply Swift destruction semantics to C++ classes. This approach will work for most C++ classes. However, C++ classes that rely on C++ destruction semantics won't work well in Swift.

Force the user to always pick a destruction point for values of C++ types. Too much burden on the user, and the code won't look like normal Swift.

Allow the user to pick a destruction point, otherwise let the compiler pick. This approach will work, but it trusts the user to do the right thing every time a certain C++ class is used. Often, whether precise destruction point is required or not depends on the type.

Define a destruction point for C++ classes. This can't be done universally in all Swift code, but it is feasible for local variables of concrete types.

An example where it is not possible to maintain precise destruction semantics for C++ classes is in a generic Swift function that is called with a C++ class as a type parameter. In this case, we must be able to compile one definition of a Swift function, which must work regardless of the nature of the type (be it a Swift type or a C++ class). This limitation does not seem too bad, because RAII types are not often passed as type parameters to templates in C++, which creates a reason to believe it will not be common in Swift either.

Allow the API owner to annotate the C++ class as requiring a precise destruction point. An improvement upon the previous approach where unusual

semantics are only applied to C++ classes that need them, for example, RAII types like mutex locks.

The pragmatic choice is to implement one of the last two options.

If a precise destruction point is defined for annotated C++ classes, here's an example of behavior that we get:

```
// C++ header.

[[swift::requires_precise_destruction]]
class PickyCxxClass {
public:
  ~PickyClass() { ... }
};
```

```
// C++ header imported in Swift.

struct PickyCxxClass {
  // The destructor is not imported as a declaration into Swift, however,
  // it is mapped to a value witness function, and it is called when the
  // lifetime of the object ends, like in C++.
}

func testConcreteType() {
  var c = PickyCxxClass()
  print("Hello")
  // `c` will be deallocated here.
}

func testTypeParameter<T>(_ t: T.self) {
  var c = T()
  // `c` can be deallocated here, even if `c` is a `PickyCxxClass`.
  print("Hello")
  // `c` can also be deallocated here.
}

testTypeParameter(PickyCxxClass.self)
```

Member functions that return inner pointers

When a member function in a C++ class returns a pointer (or a reference), most commonly the lifetime of that pointer is limited to the lifetime of the object itself. C++ code often relies on precise destruction semantics to ensure that the object that owns the memory lives long enough. Another problem is that the Swift compiler can move the object around in memory whenever it wants, invalidating those outstanding inner pointers.

```
// C++ header.
class Employee {
private:
  std::string company_;
public:
  std::string *mutable_company() { return &company_; }
};
```

```
// C++ header imported in Swift.

struct Employee {
  func mutable_company() -> UnsafeMutablePointer<std.string>
}

func test() {
  var employee: Employee = ...
  var company = employee.mutable_company()

  // HAZARD: `employee` could have been moved to a different location in memory
  // by now, invalidating the `company` pointer.
  company.pointee = "A"

  print(employee)

  // HAZARD: `employee` can be deallocated here, however, `company` is still
  // accessible.
  company.pointee = "B"
}
```

To fix the problem of values being moved around in memory, we need to tell the Swift compiler about the connection between pointers and objects that own the memory.

```
// C++ header.

class Employee {
private:
  std::string company_;
public:
  [[swift::returns_inner_pointer]]
  std::string *mutable_company() const { return &_company; }
};
```

The `returns_inner_pointer` attribute can be added manually to the C++ headers, or it can be safely inferred on all member functions that return a pointer or reference, to reduce annotation burden. Inferring this annotation on functions that don't actually need it is harmless (does not change semantics, and does not create unsafety), but may prevent some optimizations.

To fix the lifetime problem, the lifetime of a local variable can be extended until the end of the scope if the current scope called a method that returns an inner pointer.

```
func test() {
  var employee: Employee = ...
  var company = employee.mutable_company()

  // ... any amount of code...

  // `employee` is destroyed at the end of the scope because of the `mutable_company` call.
}
```

This technique will not work for temporaries (instead, they need to be extended to the end of the full expression). It is also not very useful to call methods that return inner pointers on temporaries.

Differences in move semantics between C++ and Swift

Swift's "move" operation will leave the source value uninitialized (see ownership manifesto); this operation is called a "destructive move". C++ has "non-destructive moves" that leave the source value initialized, but in an indeterminate state.

The following sections address various aspects of this impedance mismatch between the two languages.

Move-only C++ classes

For the purposes of this document, we will call C++ classes that can be moved but don't have a public copy constructor and a public copy assignment operator "move-only C++ classes".

Swift does not have move-only types at the time this document was written (January 2020). There is a desire to add them, and a lot of design work has been done (see the Ownership Manifesto), however, this design has not been proposed yet.

Therefore, we will first discuss mapping move-only C++ classes to Swift before move-only types are implemented. In this situation there's clearly no good direct mapping. There are some non-ergonomic options though.

Second, we will discuss mapping move-only C++ classes to move-only types in future Swift versions.

Move-only C++ classes: mapping in Swift 5

Import move-only C++ classes as Swift structs, and statically ensure that copies are never needed.

This seems doable, although the usability and predictability of the language feature will depend on the compiler stage where the checks are done. It is easy to perform such checks at the SIL level, however, the language might not behave predictably. It is a more difficult to do such checks on the AST, but the language will be more predictable from the user's point of view. Also, AST-based checks might reject more programs than people expect.

Since value witness tables require a valid copy witness, it will be also necessary to prohibit passing such pseudo-move-only types into generic code as type parameters, or force full specialization of generic functions, and perform the checks on fully-specialized SIL.

Here's a sketch of rules for move-only types that are reasonably simple to check at compile-time and provide good diagnostics, at the expense of the resulting code being non-ergonomic. Move-only C++ classes can be imported in Swift as structs as usual, however, such structs can't be used as values. The compiler would only allow `UnsafePointers` to them, and would allow calling methods on `UnsafePointer.pointee`. When move-only types are used as C++ class members, Swift can import them as opaque blobs, and expose a `withUnsafePointer`-style API to get an `UnsafePointer` to the data member.

```
// C++ header.

// `File` is a move-only C++ class.
class File {
private:
  int file_descriptor_;
public:
  File(std::string_view filename);
  File(const File &) = delete;
  File(File &&) = default;
  ~File();

  File& operator=(const File &) = delete;
  File& operator=(File &&) = default;

  std::string ReadAll();
};

// `TwoFiles` is move-only because it is composed of move-only parts.
struct TwoFiles {
  File firstFile;
  File secondFile;
};
```

```
// C++ header imported in Swift.

struct File {
```

```
  public init(_ filename: std.string_view)
  public func ReadAll() -> std.string
}

struct TwoFiles {
  private var _firstFile: <unspecified-opaque-storage>
  private var _secondFile: <unspecified-opaque-storage>

  func withUnsafePointerToFirstFile<Result>(
    _ body: (UnsafePointer<File>)->Result
  ) -> Result {
    body(&_firstFile)
    _fixLifetime(self)
  }

  func withUnsafePointerToSecondFile<Result>(
    _ body: (UnsafePointer<File>)->Result
  ) -> Result {
    body(&_secondFile)
    _fixLifetime(self)
  }
}

// Usage example.

func useOneFile(_ f: UnsafeMutablePointer<File>) {
  // var f2 = f.pointee // compile-time error: can only call a method on 'pointee'.
  print(f.pointee.ReadAll()) // OK

  // Move `f` to a different memory location.
  var f2 = UnsafeMutablePointer<File>.allocate(capacity: 1)
  f2.moveInitialize(from: f, count: 1)
  // `f` is left uninitialized now.

  print(f2.pointee.ReadAll()) // OK
  f2.deallocate() // OK
  // The file is closed now.
}

func useTwoFiles(_ files: UnsafeMutablePointer<TwoFiles>) {
  // Like `print(files.first.ReadAll())`, if it was possible to compile it:
  files.pointee.withUnsafePointerToFirst { print($0.ReadAll()) }
}
```

If we add support for marking APIs as returning inner pointers, we could make the imported API nicer:

```
// C++ header imported in Swift.

struct TwoFiles {
  private var _firstFile: <unspecified-opaque-storage>
  private var _secondFile: <unspecified-opaque-storage>

  @_returnsInnerPointer
  var firstFile: UnsafePointer<File> { get }

  @_returnsInnerPointer
  var secondFile: UnsafePointer<File> { get }
}
```

Move-only C++ classes: mapping to move-only Swift types (in future Swift versions)

This section is based on the design for move-only Swift types that has not been officially proposed yet. See ownership manifesto for the design for ownership and move only types that we assume in this section.

Move-only C++ classes correspond to `moveonly` structs in Swift. Semantic analysis ensures that they are never copied.

Moves that destroy the source are spelled `move(x)` in Swift. Swift does not have a concept of non-destructive move; a section below discusses possible design options.

```
// C++ header.

class File {
private:
  int file_descriptor_;
public:
  File(std::string_view filename);
  File(const File &) = delete;
  File(File &&) = default;
  ~File();

  File& operator=(const File &) = delete;
  File& operator=(File &&) = default;

  std::string ReadAll();
};
```

```
// C++ header imported in Swift.

moveonly struct File {
  public init(_ filename: std.string_view)
```

```
  public deinit

  public func ReadAll() -> std.string
}

moveonly struct TwoFiles {
  var first: File
  var second: File
}

// Usage example.

func useOneFile(_ f: File) {
  print(f.ReadAll()) // OK

  // Move `f` to a different memory location.
  var f2 = move(f)

  // print(f.ReadAll()) // compile-time error: can't access `f`, its value was moved

  print(f2.ReadAll()) // OK
  endScope(f2) // OK
  // The file is closed now.
}

func useTwoFiles(_ files: TwoFiles) {
  print(files.first.ReadAll()) // OK
}
```

Non-destructive moves

Generally, Swift code that uses values of C++ class types should use Swift's destructive moves in most cases where C++ code would have used a non-destructive move. However, certain C++ APIs are designed to be used with non-destructive moves, therefore, non-destructive moves should be made available in Swift. Adding non-destructive moves to Swift as a native language feature is a non-starter.

Consider the following C++ code:

```
// Example of C++ API design that requires users to perform non-destructive
// moves.

class Example { /* move-only */ };

std::optional<Example> getExample();
void processExample(Example e);
```

```
void test() {
  std::optional<Example> e = getExample();
  processExample(std::move(e.value()));
}
```

Equivalent Swift code would look like this:

```
// Equivalent Swift code.

func test() {
  var e: cxx_std.optional<Example> = getExample();

  processExample(move(e.value()));
  // If we assume that the previous line compiled, we have a problem:
  // `cxx_std.optional` does not know that the value was destructively moved
  // and will try to delete it again.
}
```

Of course, one could argue that the interoperability layer could provide special support for `std::optional`, and make it work with destructive moves better. While that is true, the point still stands: there will be C++ APIs that expect users to perform non-destructive moves, and should ensure that they are usable from Swift without overlays or special compiler support.

Option 1: Non-destructive moves in Swift are swaps with an arbitrary valid value.

It is easy to model a non-destructive move in Swift as a swap with a default-initialized value.

```
// Swift standard library, C++ support module.

public protocol DefaultInitializable {
  init()
}

public moveonly func cxxMove<T: DefaultInitializable>(_ value: inout T) -> T {
  var result = T()
  swap(&value, &result)
  return result
}

public moveonly func cxxMove(_ value: inout T, replacingWith newValue: T) -> T {
  var result = newValue
  swap(&value, &result)
  return result
}
```

```
// Usage example.

// This function takes a `File` by value, consuming it.
func consumeFile(_ f: File) { ... }

func useArrayOfFiles() {
  var files: [File] = ...
  // consumeFile(files[0]) // compile-time error: can't copy a `File`

  // Swift's move does not help.
  // consumeFile(move(file[0]))
  // compile-time error: moving one element of an array will cause a
  // double-destruction of `File` when the array is destroyed.

  consumeFile(cxxMove(&file[0])) // OK, replaced `file[0]` with a `File()`.

  consumeFile(cxxMove(&file[1], replacingWith: File("/tmp/example.txt")))
  // OK, replaced `file[1]` with a newly-opened file.
}
```

The advantage of this approach is that we avoid creating "valid but indeterminate" values, which can be hazardous. However, such values can be passed from C++ to Swift at any time, even if Swift code can't create them directly. It is also awkward to require the user to come up with a dummy valid value when the type is not default-constructible.

Option 2: Non-destructive moves in Swift invoke the C++ move constructor.

So, we want to perform a real C++ move and leave a "valid but indeterminate" value in the source. For that, we must enable the Swift to call the C++ move constructor.

If `class File` has a move constructor in C++, the imported Swift struct defines a corresponding initializer:

```
// C++ header imported in Swift.

public moveonly struct File {
  // File(File &&);
  public init(cxxMovingFrom: inout File)
}

// Usage example.

func useArrayOfFiles() {
  var files: [File] = ...
  consumeFile(File(cxxMovingFrom: &files[0]))
}
```

The move syntax, `File(cxxMovingFrom: &files[0])`, is quite verbose and requires repeating the name of the type, whereas the corresponding C++ construct, `std::move(x)`, does not. When type context is available, the expression can be abbreviated to `.init(cxxMovingFrom: &files[0])`. We can make it more concise by synthesizing a helper method in movable types:

```
// C++ header imported in Swift.

public moveonly struct File {
  // File(File &&);
  public init(cxxMovingFrom: inout File)

  public mutating func cxxMove() -> File {
    return File(cxxMovingFrom: &self)
  }
}

// Usage example.

func useArrayOfFiles() {
  var files: [File] = ...
  consumeFile(files[0].cxxMove())
}
```

Although methods are more in line with Swift API design than free functions, injecting names into C++ types can backfire due to naming collisions.

Alternatively, we could model non-destructive moves with a free function, imitating the C++ syntax. To do this we need a protocol for non-destructively movable types, and a free function that calls the move constructor:

```
// Swift standard library, C++ support module.

public protocol NonDestructivelyMovable {
  public init(cxxMovingFrom: inout Self)
}

public moveonly func cxxMove<T: NonDestructivelyMovable>(_ value: inout T) -> T {
  return T(cxxMovingFrom: &value)
}

// C++ header imported in Swift.

public moveonly struct File: NonDestructivelyMovable {
  public init(cxxMovingFrom: inout File)
}

// Usage example.
```

```
func useArrayOfFiles() {
  var files: [File] = ...
  consumeFile(cxxMove(&files[0]))
}
```

It is also useful to expose the move assignment operation in Swift. Even though move assignment is not necessary to implement `cxxMove()`, it can be used for optimization.

```
// C++ header imported in Swift.

public moveonly struct File {
  // File(File &&);
  public init(cxxMovingFrom: inout File)

  // File& operator=(File &&);
  public func moveAssignFrom(_ value: inout File)
}
```

The compiler could perform a high-level optimization replacing `x = cxxMove(&y)` (which creates a temporary, and calls the move constructor twice) with `x.moveAssignFrom(&y)` (which does not create a temporary and calls move assignment only once).

Non-movable C++ classes

By non-movable C++ classes we mean C++ classes that don't have a copy constructor or a move constructor.

Swift does not have any struct-like equivalent to non-movable C++ classes, and there's nothing planned in that area. Therefore, we have to either creatively reuse one of existing Swift's features, or add some new features to Swift.

Option 1: Map non-movable C++ classes to Swift structs.

If we want to map all C++ classes to structs in Swift, one approach to make non-movable C++ classes work is only importing pointers to non-movable types; this approach is described in detail in the section about importing move-only types in Swift 5. Non-movable classes in C++ are often used as polymorphic base classes. Allowing to only use pointers to such types is not a bad limitation. The primary disadvantage of this approach is ergonomics.

Option 2: Map non-movable C++ classes to Swift classes.

While Swift structs are a bad match for non-movable C++ classes, Swift classes match semantics better. For example, instances of Swift classes, once allocated, stay at the same address until `deinit`. Instances of Swift clasess are always worked with indirectly, eliminating the need to move them around in memory.

Although user-defined Swift classes have a certain memory layout determined by

the Swift compiler (for example, the memory block should start with an object header that contains a metadata pointer and a refcount), that exact memory layout is not necessary for a type to successfully masquerade as a Swift class. For example, Swift already imports CoreFoundation types as "foreign classes"; from the user's point of view they look like ordinary classes.

The biggest semantic impedance mismatch between C++ classes and Swift classes is that Swift classes are reference-counted. It is certainly possible to import a C++ class as a Swift class if it is intrusively reference counted, or wrapped in a `shared_ptr`.

But what about non-movable C++ classes that are not reference counted? This issue could be worked around by making retain and release operations on C++ classes be a no-op, and introducing an explicit "delete" operation, effectively making C++ class references in Swift equivalent to `UnsafeMutablePointer`.

```
// C++ header.

class NonMovable {
public:
  NonMovable(const NonMovable &) = delete;
  NonMovable(NonMovable &&) = delete;
  ~NonMovable();

  NonMovable &operator=(const NonMovable &) = delete;
  NonMovable &operator=(NonMovable &&) = delete;

  void debugPrint() const;
};
```

```
// C++ header imported in Swift.

class NonMovable {
  // Runs `delete this`.
  func delete()

  func debugPrint()
}

func useNonMovable(nm: NonMovable) {
  var nm2 = nm
  // `nm2` now points to the same object as `nm`. No reference-counting.

  nm.debugPrint() // OK
  nm2.debugPrint() // OK

  nm2.delete() // OK
  // The object is deallocated now. From this point, using `nm` or `nm2`
```

```
  // is undefined behavior.
}
```

Everything seems to work really well, except that we broke Swift's safety guarantees about classes. Swift manages lifetime of regular user-defined classes automatically; in safe Swift code it is not possible to get a dangling class reference; it is also weird that users would have to manually deallocate class instances. Therefore, classes imported from C++ would behave very differently from user-defined Swift classes; most importantly, they are unsafe. This issue seems incompatible with Swift's design and philosophy. Otherwise, importing C++ classes as Swift classes looks like a viable implementation strategy.

The unsafety issue could be mitigated by adding the word "unsafe" somewhere in the source code. For example, importing a non-movable C++ class `Example` as a Swift class `UnsafeExample`, or requiring the user to write the keyword "unsafe" before the type name:

```
func useNonMovable(nm: unsafe NonMovable) { ... }
```

Member functions

Member functions of C++ classes are most naturally mapped to methods in Swift structs. C++ allows to add various qualifiers to member functions, which we discuss in the following sections.

Const-qualified member functions in C++ classes

Member functions of C++ classes can be marked with a `const` qualifier; however, this qualifier does not provide any semantic guarantees. Const-qualified methods can change the state of the object through a variety of mechanisms.

`const_cast` is the first language feature that comes to mind when one thinks about modifying a const object. As long as an object was originally allocated as non-const, its const-qualified methods can cast const away and mutate the object. Detecting potential `const_cast`s is quite difficult, since it requires examining everything that a const member function can call, transitively. However, in practice, `const_cast`s are rare, and given the more pervasive issues described below, `const_cast`s can be ignored.

```
class SneakyMutation {
private:
  int value = 0;
public:
  void mutateSneakily() const { opaqueMutator(this); }
  void setValue(int newValue) { value = newValue; }
};

// in a different file:
void opaqueMutator(const SneakyMutation* sm) {
```

```
  const_cast<SneakyMutation*>(sm)->setValue(42);
}
```

Another C++ feature that allows const-qualified member functions to mutate the object is "mutable" data members. Mutable data members are a lot more common than `const_cast`s, but fortunately they are trivial to detect statically.

```
class SneakyMutation {
private:
  mutable int value = 0;
public:
  void mutateSneakily() const { value = 42; }
};
```

Yet another C++ feature that allows const-qualified methods to mutate the object is aliasing. A const member function can obtain a pointer to the same object through some other means, and change the pointed-to data.

```
class SneakyMutation {
private:
  int value = 0;
public:
  void mutateSneakily(SneakyMutation *other) const {
    other->value = this->value + 1;
    // What if `other` happens to be equal to `this`? In that case, a const
    // member function has mutated `this`!
  }
};

void test() {
  SneakyMutation sm;

  // A const method mutates the object that it is invoked on:
  sm.mutateSneakily(&sm);
}
```

Note that `mutateSneakily()` can obtain the alternative `this` pointer through a variety of ways, not necessarily as a function argument. The method can find another non-const pointer equal to `this` in a global variable or in a data structure accessible from `this` itself. It also need not mutate `other` so obviously: it can pass `other` to another function which will mutate it. It is difficult to estimate how common such mutation is, however, it is not an element of a common design pattern.

Swift's replacement for C++ const member functions are non-`mutating` functions:

```
struct Example {
  private var value: Int = 0
```

```
  mutating func mutatingFunction() { value = 42 }
  func nonMutatingFunction() {
    // value = 42 // would not compile
    print(value) // OK
  }
}
```

The biggest difference from C++ is that non-`mutating` functions in Swift are really not allowed to mutate `self`; there are no backdoors. Some equivalents of C++'s mutation backdoors are disallowed statically, some are disallowed dynamically, some are undefined behavior.

The backdoors are disallowed by the exclusivity rule (from SE-0176 Enforce Exclusive Access to Memory):

> two accesses to the same variable are not allowed to overlap unless both accesses are reads

If we map C++ classes to Swift structs, how do we map C++'s `const` qualifier on member functions? There are a couple of options.

- Ignore `const`, and mark all Swift struct functions as `mutating`. Allow API owners to manually annotate functions as non-`mutating`. This is a conservative and correct-by-default approach, but it is not ergonomic.
- Map `const` to non-`mutating` in Swift. This approach is very desirable from the point of view of ergonomics, however it is not safe in the general case. Nevertheless, in many cases such inference will be correct.

How can we make the second, ergonomic, approach safer? First of all, we should allow C++ API owners to override the inferred `mutating`/non-`mutating` with an annotation. API owners must have control over what their APIs look like in Swift.

In addition, the compiler could do a lightweight static analysis to recognize common problematic patterns, like `mutable` members and obvious `const_cast`s. If the compiler finds anything alarming, it should refuse to perform inference and require the user to manually annotate methods in C++ as `mutating` or non-`mutating`.

Volatile-qualified member functions

Volatile-qualified member functions are so rare that there's not enough value in spending non-trivial engineering effort for them in the interop layer ensuring that they map nicely. There are two easy ways to handle them:

- don't import volatile-qualified member functions,
- map member functions as if the `volatile` qualifier was not present.

Ref-qualified member functions

TODO

Final functions

TODO

Getters and setters

Getters and setters in C++ classes most directly correspond to properties in Swift. While it is certainly possible to map getters and setters to regular methods, properties would be more ergonomic.

First of all, we need to find getter/setter pairs in a C++ class. We would need to use name-based heuristics for that, while allowing the API owner to override the inference with annotations (specifically, with the `swift_name` attribute). There are a variety of naming styles for getters and setters in C++, but we only need to cover the most common ones.

Getters take no arguments, return a `T` or a `const T`. Typical naming patterns are: `AaaBbb()`, `GetAaaBbb()`, `aaaBbb()`, `getAaaBbb()`, `aaa_bbb()`, `get_aaa_bbb()`

Setters take a `T`, a `const T&`, or a `T&&`, and return `void`. Typical naming patterns are: `SetAaaBbb()`, `setAaaBbb()`, `set_aaa_bbb()`

Mutable accessors take no arguments and return a `T&` or a `T*`. Typical naming patterns are: `MutableAaaBbb()`, `mutableAaaBbb()`, `mutable_aaa_bbb()`.

The C++ importer can synthesize a Swift property from a getter, from a getter/setter pair, or from a getter and a mutable accessor. Swift does not have set-only properties, so unpaired setters and mutable accessors will be imported as regular methods.

```
// C++ header.

class Employee {
public:
  const std::string &getName() const;
  void setName(std::string newName);
};
```

```
// C++ header imported in Swift.

struct Employee {
  public var name: std.string { get set }
}
```

```
// Implementation details of the imported API.

struct Employee {
```

```
  // const std::string &getName();
  private func _getName() -> UnsafePointer<std.string>

  // void setName(std::string newName);
  private mutating func _setName(_ newName: std.string)

  // Swifty API.
  public var name: std.string {
    _read {
      yield _getName().pointee
    }
    set {
      _setName(newValue)
    }
  }
}
```

Sometimes the type returned by the getter will not match the type taken by the setter. For example, the getter can return a `std::string_view`, while a setter takes a `std::string`. `std::string_view` is like a `const std::string&`, so it makes sense from the API perspective in C++. Swift, however, requires the getter and setter to use the same type exactly.

We could teach the C++ importer about the most widely used type pairs, and insert the appropriate type conversions in the synthesized property accessors. However, these type conversions will carry a performance penalty, and the resulting Swift API will be less performant than the original C++ API. For example, consider a C++ class where a getter returns a `std::string_view`, and the setter takes a `std::string`. If we synthesize a `std::string_view` property in Swift, its setter would always copy the character data, even though in C++ the setter would move the `std::string`. If we synthesize a `std::string` property in Swift, its getter would have to materialize a `std::string` from a `std::string_view`.

Therefore, it seems like the best approach is to teach the C++ importer about handling such mismatched types, but only perform property synthesis on demand, triggered by an annotation in C++ code.

```
// C++ header.

// New attribute.
#define SWIFT_PROPERTY(type, name) __attribute__((swift_property(#type, #name)))

class Manager {
public:
  std::string_view getName() const SWIFT_PROPERTY(std::string, name);
  void setName(std::string newName) SWIFT_PROPERTY(std::string, name);
```

```
  std::span<const Employee> getReports() const;
  void setReports(std::vector<Employee> newReports);
};

// C++ header imported in Swift.

struct Manager {
  // Property is synthesized as requested by the annotation.
  public var name: std.string { get set }

  // Property is not synthesized because getter and setter operate on
  // different types.
  public func getReports() -> std.span<const<Employee>>
  public func setReports(_ newReports: std.vector<Employee>)
};
```

Pointers to members

Pointers to members are quite rare in C++, so not importing them at all, at least initially, would not be a big loss. Nevertheless, importing them is possible.

Pointers to data members best correspond to keypaths in Swift. (TODO: need an example.)

Pointers to member functions could be ergonomically mapped to curried functions, just like struct and class methods in Swift can be converted to curried functions.

```
// Inspiration from Swift: methods can be converted to curried functions.

// A normal user-defined struct.
struct Point {
  var x: Double
  var y: Double

  func distanceTo(_ line: Line) -> Double { ... }
}

var distanceToFunc: (Point) -> (Line) -> Double = Point.distanceTo
```

The only difference is that C++ member function pointers use a different memory layout than Swift closures. Therefore, they would need to be imported with a special calling convention, let's call it `@convention(cxx_method)`. Here's an example:

```
// C++ header.

class Point {
public:
  double distanceTo(Line line) const;
```

```
};

void CallMemberFunctionPointer(Point p, double (Point::*ptr)(Line));

// C++ header imported in Swift.

public struct Point {
  public func distanceTo(_ line: Line) -> Double { ... }
}

func CallMemberFunctionPointer(
  _ p: Point,
  _ ptr: @convention(cxx_method) (Point) -> (Line) -> Double
)
```

Virtual member functions

If we map C++ classes to Swift structs, it makes the most sense to map virtual member functions to struct methods in Swift. However, Swift structs don't support inheritance. So, while we can map virtual functions to methods and execute C++ dynamic dispatch correctly, we can't express the fact that the struct is subclassable, and that the method is overridable in Swift.

TODO: design how to map inheritance.

Inherited APIs

Swift structs don't allow inheritance. If we map C++ classes as Swift structs, we could replicate APIs from all base classes in every derived class.

```
// C++ header.

class Base {
public:
  void doBaseStuff();
};

class Derived : public Base {
public:
  void doDerivedStuff();
};

// C++ header imported in Swift.

public struct Base {
  public func doBaseStuff()
}
```

```
public struct Derived {
  public func doBaseStuff()
  public func doDerivedStuff()
}
```

Conversions between derived and base classes

Swift structs don't allow inheritance, so it is not possible to directly express the is-a relationship between derived classes and base classes. The is-a relationship is not very important for non-polymorphic C++ classes, where inheritance is typically used as an implementation technique, rather than an intentional part of the API. In polymorphic classes the is-a relationship is quite important and is an intentional part of the API.

So how to represent conversions between derived and base classes? There are two cases to consider: converting values and converting pointers.

Converting values of derived type to values of base type is known as "object slicing" in C++. It is well-known to be error-prone, but since some C++ APIs might rely on it, we should try to expose it in Swift.

Object slicing is possible without any language extensions:

```
var derived: Derived = ...
var base: Base = UnsafeRawPointer(&derived).load(as: Base.self)
```

It is not very ergonomic, but not too bad for an operation that is known to be error-prone. APIs that intentionally rely on object slicing could add helper methods to types that intentionally participate in slicing:

```
// Manually written overlay for a C++ API.

extension Base {
  public init(_ other: Derived) {
    self = UnsafeRawPointer(&other).load(as: Base.self)
  }
  public init(_ other: UnsafePointer<Derived>) {
    self =  UnsafeRawPointer(other).load(as: Base.self)
  }
}
```

Now let's talk about conversions between pointers to derived type and pointers to base type. These operations need to be ergonomic only for polymorphic classes.

First of all, let's reject extending Swift's type casting operators `as` and `as?` to handle casts between pointers to C++ objects. We need to convert between `UnsafePointer<Base>` and `UnsafePointer<Derived>`, which are not connected with a subtyping relationship in Swift. Implementing a conversion in type casting operators (`as` and `as?`) that does not reflect a subtyping relationship is not a good idea. Swift already has a couple of such conversions, and they cause a

lot of complexity in the compiler and the runtime. Moreover, sometimes such casting behavior can be unexpected by users. Therefore, we are going to explore API-based approaches to express upcasts and downcasts for C++ objects.

It is important to realize that interleaved accesses to an `UnsafePointer<Base>` that aliases an `UnsafePointer<Derived>` violate Swift's type-based aliasing rules. These rules could be relaxed for types imported from C++.

To prevent the user from accidentally violating type-based aliasing rules, Swift intentionally does not provide an easy way to freely convert between pointers of unrelated types. Swift allows to temporarily create an aliasing pointer; however, that pointer has a scoped lifetime, and during that scope accessing the original pointer is not allowed.

```
// Pointer conversion example in existing Swift language.

var basePtr: UnsafePointer<Base> = ...
basePtr.withMemoryRebound(to: Derived.self) {
  // In this closure, `$0` has type `UnsafePointer<Derived>` and
  // aliases `basePtr`.
  $0.doBaseStuff()
  $0.doDerivedStuff()
}
```

Type conversions are usually expressed as initializers in Swift. However, adding an initializer to `UnsafePointer` for base/derived C++ class pair is going to create a big overload set, which will negatively impact compilation performance. Therefore, we should try to distribute the conversion APIs among the imported types themselves. The importer could synthesize conversion APIs on base and derived classes:

```
// C++ header imported in Swift.

public struct Derived {
  @returnsInnerPointer
  public var asBase: UnsafePointer<Base> { get }

  public static func downcast(from basePointer: UnsafePointer<Base>) -> UnsafePointer<Derive
}

public struct Base {
  public static func upcast(from basePointer: UnsafePointer<Derived>) -> UnsafePointer<Base>
}

// Usage example.

func useBasePtr(_ basePtr: UnsafePointer<Base>) { ... }
func useDerivedPtr(_ derivedPtr: UnsafePointer<Derived>) { ... }
```

```
func testDowncast() {
  var d = Derived()
  useBasePtr(d.asBase)

  // or:
  useBasePtr(Base.upcast(from: &d))
}

func testUpcast(_ basePtr: UnsafePointer<Base>) {
  if let derivedPtr: UnsafePointer<Derived> = Derived.downcast(from: basePtr) {
    useDerivedPtr(derivedPtr)
  }
}
```

The `asBase` property operates on a value, so it will need a way to pin the original value in memory while the aliasing pointer is live. If we ignore that API, the "type-upcast-from-value" and "type-downcast-from-value" syntax in other APIs doesn't read well, because the order looks reversed.

Let's see what an API based on free functions would look like. We have to separate the upcast and downcast APIs because they must return different types: upcasts can't fail, while downcasts can.

```
// C++ support module in the Swift standard library.

public func cxxUpcast<Source, Destination>(
  _ source: UnsafePointer<Source>,
  to type: Destination.Type
) -> UnsafePointer<Destination>

public func cxxDowncast<Source, Destination>(
  _ source: UnsafePointer<Source>,
  to type: Destination.Type
) -> UnsafePointer<Destination>

public func cxxDynamicDowncast<Source, Destination>(
  _ source: UnsafePointer<Source>,
  to type: Destination.Type
) -> UnsafePointer<Destination>?
```

The Swift type checker would have special knowledge about these functions, and only allow calling them when `Source` and `Destination` are fully concrete types that belong to the same class hierarchy in C++.

These functions will know the concrete source and destination types, so they will be able to perform any necessary pointer adjustments, therefore multiple inheritance and virtual base classes wouldn't be a problem.

Subclassing C++ classes in Swift

TODO

Making C++ classes conform to Swift protocols

TODO

Enums

In Swift 5, C enums that are not annotated (with either of `enum_extensibility`, `NS_ENUM`, or `NS_OPTIONS`) are imported into Swift as structs, and enumerators are imported as computed global variables. This might have been the right call for importing Objective-C code, where enums not declared with `NS_ENUM` are highly likely to have a "non-enum nature": they are either a bitfield, or just a bag of constants. C++ has more ways, and more flexible ways to declare constants, so using enums in C++ to declare constants is not very idiomatic. Bitfield enums are used in C++ though, however, compared to overall enum usage, bitfield enums are still rare. Therefore, current strategy of importing non-annotated enums as Swift structs is the wrong default for C++.

There are multiple strategies we could take to import C++ enums into Swift.

(1) Adopt the strategy used for C enums in C++: unannotated C++ enums are imported as Swift structs, C++ enums annotated as open/closed are imported as Swift enums.

Pros:

- Conservatively correct. This strategy does not trust enums to have the "enum nature" unless they are annotated otherwise.
- Simpler design: identical rules across C, Objective-C, and C++.

Cons:

- Non-ergonomic. In idiomatic C++ codebases most enums have enum nature, so users would need to add lots of annotations.

(2) Assume non-annotated C++ enums have "enum nature" and import them as non-frozen Swift enums; (optionally) implement an annotation for C++ enums with "non-enum nature" and import such enums as Swift structs.

Pros:

- Correct default. In an idiomatic C++ codebase, the vast majority of enums do have enum nature.
- Maintaining source compatibility by not changing how C enums are imported.

Cons:

- Non-annotated enums of non-enum nature would be imported incorrectly from the API point of view (however, there is no undefined behavior hazard).
- No clear strategy to tell apart C and C++ enums. In an ideal world, `extern "C"` should allow to detect C declarations in C++, but not all C code is wrapped in `extern "C"`. For example, most Objective-C headers are not wrapped in `extern "C"` because most Objective-C constructs have a compatible ABI between Objective-C and Objective-C++.

(3) Apply strategy #2 in C, Objective-C, and C++. Doing so will change the mapping strategy for non-annotated enums coming from Objective-C code, breaking source stability to some extent.

Pros:

- Correct default in C++.
- Likely correct default in C.
- Simpler design: same rules in C and C++.
- Obvious how to implement, because there is no need to distinguish between different header kinds.

Cons:

- Likely not correct default in Objective-C. In idiomatic Objective-C code enums with "enum nature" are declared with `NS_ENUM`, and bitfields are declared with `NS_OPTIONS`, which leaves plain enums to cover the unusual cases.
- Breaking source compatibility by changing how C enums are imported. There are ways to mitigate this issue, for example, continue importing enumerators as computed global variables, marking them deprecated.

Strategy #3 is best for C++ code, but it is unclear if consequences for Objective-C would be acceptable. We should gather data about usage of plain enums in idiomatic C++ code and in Apple's SDKs and evaluate the absolute amount of source breakage.

Templates

From 10,000 feet, C++ templates are similar to Swift generics. However, there are multiple semantic gaps between them.

C++ templates perform a syntactic substitution. Any type that supports the syntax invoked in the template is a valid template argument. However, we don't know exactly what syntax a type is required to support until we try to substitute that type into a template. Generics in Swift are constraint-based and require modular type checking.

C++ templates require instantiation at compile time. It is not possible to compile a template without substituting concrete types in. It is not specified whether Swift generics are instantiated or not, the compiler can make either choice and the user can't tell the difference.

C++ templates support specialization. Swift generics don't allow specialization; they provide a different mechanism (protocol requirements) to make one generic function behave differently for different types.

C++ class templates support specialization, which allows defining completely different APIs across specializations of the same class. All instantiations of a generic type in Swift have the API described in the generic declaration (with type parameters substituted), plus applicable conditional extensions.

C++ templates support non-type template parameters, template template parameters, and parameter packs (variadic generics), all of which are not supported in Swift.

Function templates

Since C++ uses an instantiation-based model for templates, the most obvious way to import C++ function templates in Swift is to only import full specializations as regular functions. However, it would be most ergonomic to import function templates as Swift generic functions. Let's explore both options and see how far we can go.

Function templates: import as Swift generic functions

Let's take explicit specializations out of the picture for the purposes of presenting the API in Swift. Since overload resolution in C++ ignores specializations of function templates, we can ignore them, too, when importing a function template into Swift.

```
// C++ header.

template<typename T>
void functionTemplate(T t) { ... }

// Any number of explicit specializations may follow.
template<>
void functionTemplate(int x) { ... }
```

```
// C++ header imported in Swift.

func functionTemplate<T>(_ t: T)

// No need to import explicit specializations as separate APIs.
```

Of course, when Swift code calls `functionTemplate` with an `Int32`, it should call the correct explicit specialization, `functionTemplate<int>`, even though it is not shown in the imported Swift API.

Function templates: allow to specify template arguments

Calls to Swift generic functions always deduce generic arguments from the call. C++ allows the caller to either deduce the arguments, or specify them explicitly. Sometimes it is important to invoke C++ function templates with specific template arguments, so it would make sense to expose this capability in Swift.

We suggest to allow Swift code to use the angle bracket syntax to explicitly specify function template arguments at the callsite. If template arguments are not specified, we should run template argument deduction. (TODO: explain in more detail how template argument deduction will work starting from Swift types.)

```
// Usage example in Swift.

functionTemplate<Int>(42)
```

How to specify C++ types that map ambiguously into Swift, for example, `int &` or `int *`, both of which map to `UnsafePointer<Int>`? The C++ support module in the Swift standard library could provide Swift typealiases that would map unambiguously to C++.

```
// C++ support module in the Swift standard library.

typealias CxxPointer<T> = UnsafeMutablePointer<T>      // T*
typealias CxxConstPointer<T> = UnsafePointer<T>        // const T*

typealias CxxRef<T> = UnsafeMutablePointer<T>          // T&
typealias CxxConstRef<T> = UnsafePointer<T>            // const T&

typealias CxxRvalueRef<T> = UnsafeMutablePointer<T>    // T&&
typealias CxxConstRvalueRef<T> = UnsafePointer<T>      // const T&&

// Usage example in Swift.

var x = 42

// Calls `functionTemplate<int>`.
functionTemplate<Int>(x)

// Calls `functionTemplate<const int *>`.
functionTemplate<CxxConstPointer<Int>>(&x)

// Calls `functionTemplate<int &>`.
```

```
functionTemplate<CxxRef<Int>>(&x)
```

Although it is desirable to map rvalue references to Swift's `inout`, it is not possible to do so when importing a C++ function template as a C++ generic function; "inout"-ness of an argument can't change across instantiations of a generic function in Swift.

Function templates: calls to specific specializations

From an implementation point of view, it is easy to compile Swift code that calls C++ function templates if template arguments are concrete types that are obvious at the callsite (either deduced or specified explicitly). In other words, it is easy to compile the call if Swift compiler can easily deduce which specialization of the C++ function template to call. The imported API could still look like a Swift generic function:

```
// C++ header imported in Swift.

func functionTemplate<T>(_ t: T)

// No need to import explicit specializations as separate APIs.

// Usage example in Swift, works.

func callFunctionTemplate() {
  functionTemplate(0) // OK: calls the `functionTemplate<int>` specialization.
}
```

If we stop here, `functionTemplate` would look like a generic function in Swift, but it would not be callable from generic code. This option might be good enough for the most basic mapping of function templates in Swift. But can we allow calling C++ function templates from Swift generics?

Function templates: calls with generic type parameters

The primary implementation difficulty in allowing generic Swift code to call C++ function templates is that the C++ compiler can only compile full specializations, but Swift generics can run unspecialized.

```
// Usage example in Swift.

func myGenericFunction<T>(_ t: T) {
  functionTemplate(0) // OK: calls the `functionTemplate<int>` specialization.

  functionTemplate(t) // compilation error: can't call a C++ function template, because we
}
```

We could force Swift generics to be instantiated if they use a C++ template. Fully generic entry points will be still emitted to fulfill ABI expectations, but

they would trap.

```
// C++ header imported in Swift.

@_must_specialize
func functionTemplate<T>(_ t: T)

// Usage example in Swift.

@_must_specialize
func genericFunction<T>(_ t: T) {
  functionTemplate<T>(t) // OK!
}

struct GenericStruct<T> {
  init GenericStruct(_ t: T) { self.t = t }
  var t: T

  @_must_specialize
  func callGenericFunction() {
    genericFunction(t) // OK!
  }
}

func example() {
  GenericStruct(42).callGenericFunction()
}
```

We introduce a `@_must_specialize` attribute for generic functions that must be instantiated in order to be called. The importer marks all Swift generic functions that are backed by a C++ template as `@_must_specialize`, and from there the Swift compiler infers the attribute for all users.

This analysis relies on over-approximating the dynamic callgraph with the static callgraph. The static callgraph is feasible to compute in most cases, since Swift has very limited ways to abstract over an unspecialized generic function. Speicifically, Swift does not have generic closures. Swift does allow protocols to have requirements for generic functions though.

Function templates: importing as real generic functions

If we know the complete set of allowed type arguments to a C++ function template, we could import it as an actual Swift generic function that performs dynamic dispatch.

```
// C++ header.

template<typename T>
```

```
void functionTemplate(T t) { ... }

// Any number of explicit specializations may follow.
template<>
void functionTemplate(int x) { ... }
```

Let's say it is known that `functionTemplate` can only take `Int` and `UInt`.

```
// C++ header imported in Swift.

func _cxx_functionTemplate<T>(_ t: T) // Imported as explained before.

func functionTemplate<T>(_ t: T) {
  if T == Int.self {
    return _cxx_functionTemplate(t as! Int)
  }
  if T == UInt.self {
    return _cxx_functionTemplate(t as! UInt)
  }
  fatalError()
}
```

The user can write this code themselves (in an overlay, for example), or if we can communicate the allowed list of types to the importer, it can synthesize the function for us. Either way, the result is that we get a true generic Swift function.

Class templates

Class templates pose the same challenges as function templates regarding separate compilation. In addition to that, we can't ignore explicit specializations of class templates when importing the API to Swift. We could ignore explicit specializations of function templates, because they don't affect the API. Explicit specializations of class templates can dramatically change the API of the type.

Class templates: Importing full class template instantiations

A class template instantiation could be imported as a struct named `__CxxTemplateInst` plus Itanium mangled type of the instantiation (see the `type` production in the Itanium specification). Note that Itanium mangling is used on all platforms, regardless of the ABI of the C++ toolchain, to ensure that the mangled name is a valid Swift type name (this is not the case for MSVC mangled names). A prefix with a double underscore (to ensure we have a reserved C++ identifier) is added to limit the possibility for conflicts with names of user-defined structs. The struct is notionally defined in the `__C` module, similarly to regular C and C++ structs and classes. Consider the following C++ module:

```
// C++ header.

template<class T>
struct MagicWrapper {
  T t;
};
struct MagicNumber {};

typedef MagicWrapper<MagicNumber> WrappedMagicNumber;
```

`WrappedMagicNumber` will be imported as a typealias for a struct `__CxxTemplateInst12MagicWrapperI11MagicNumberE`. Interface of the imported module will look as follows:

```
// C++ header imported to Swift.

struct __CxxTemplateInst12MagicWrapperI11MagicNumberE {
    var t: MagicNumber
}
struct MagicNumber {}
typealias WrappedMagicNumber = __CxxTemplateInst12MagicWrapperI11MagicNumberE
```

Class templates: importing specific specializations

Just like with calls to C++ function templates, it is easy to compile a use of a C++ class templates if the usage in Swift code unambiguously specifies which specialization should be used.

```
// C++ header.

template<typename T>
class ClassTemplate {};

// C++ header imported to Swift.

struct ClassTemplate<T> {}

// Usage example.

func useClassTemplate() {
  var x = ClassTemplate<Int32>() // OK, uses `ClassTemplate<int>`.
}
```

If the class template provides specializations with completely different APIs, we don't have an issue, because we only need to import specific specializations into Swift.

```
// C++ header.
```

```
template<typename T>
class ClassTemplate {
  void func1();
};

template<>
class ClassTemplate<int> {
  void func2();
};

template<typename T>
class ClassTemplate<std::vector<T>> {
  void func3();
};
```

```
// C++ header imported to Swift.

// A shell of a generic struct only used for name lookup.
struct ClassTemplate<T> {}
```

```
// Usage example.

func useClassTemplate() {
  var x1 = ClassTemplate<Int8>() // OK
  var x2 = ClassTemplate<Int32>() // OK
  var x3 = ClassTemplate<cxx_std.vector<Int32>> // OK
}
```

When the Swift compiler sees `ClassTemplate<Int8>`, it translates the type to C++, that is, `ClassTemplate<char>`. Then Swift compiler asks the Clang importer to import that specialization. Clang importer instantiates the primary template with `T=char` and imports the resulting `ClassTemplate<char>`. Same for all other rereferences to `ClassTemplate`. At the end, we get:

```
// C++ header imported to Swift, as needed for the program above.

// A shell of a generic struct only used for name lookup.
struct ClassTemplate<T> {}

struct ClassTemplate<Int8> {
  func func1()
}

struct ClassTemplate<Int32> {
  func func2()
}

struct ClassTemplate<cxx_std.vector<Int32>> {
```

```
  func func3()
}
```

Effectively, `ClassTemplate<Int8>`, `ClassTemplate<Int32>`, and `ClassTemplate<cxx_std.vector<Int32>>` are completely unrelated types; they just have angle brackets in their name. (By the way, the same is the case in C++.)

Advantages of this approach:

- Relatively straightforward to implement.
- Simple model, few or no corner cases.

Disadvantages of this approach:

- Users can only use complete specializations. In other words, users can't use a C++ template from Swift generic and pass a generic type parameter to the C++ template.

Class templates: using with generic type parameters

Many class templates don't define specializations that dramatically change the API, or use features like non-type template parameters that don't exist in Swift generics. It is desirable to import such templates as Swift generics directly. With a lightweight static analysis, or with annotations in the C++ header, the compiler would recognize such a type and import the API shared by all instantiations into Swift.

Similarly to our treatment of function templates described above, we can annotate the transitive closure of generic Swift code that uses C++ class templates with `@_must_specialize`.

Class templates: using in generic code through a synthesized protocol

Class templates that have a uniform API across all specializations could be made to conform to a Swift protocol, which would facilitate passing them to generic code.

```
// C++ header.

template<typename T>
class MyClassTemplate {
  void func1(T t);
};
```

```
// C++ header imported in Swift.

protocol MyClassTemplateProtocol {
  associatedtype T
  func func1(_ t: T)
}
```

```
struct MyClassTemplate<T> : MyClassTemplateProtocol {}
```

In this model, the Swift compiler can still import only specific specializations of `MyClassTemplate`. The new part is that the Swift compiler will synthesize a protocol, `MyClassTemplateProtocol`, that contains APIs shared by all specializations. The Swift compiler will also make every imported specialization of `MyClassTemplate` conform to `MyClassTemplateProtocol`.

This way, we solved the issue of specializations being unrelated types: now they all conform to a common protocol, and therefore, generic code can operate transparently on any specialization.

```
// Usage example.

// Use a C++ class template from a Swift generic function without naming a
// specific specialization.
func useGeneric<SomeSpecialization>(_ classTemplate: SomeSpecialization, _ t: SomeSpecializa
  where SomeSpecialization: MyClassTemplateProtocol
{
  classTemplate.func1(t)
}

func useConcrete() {
  var classTemplate = MyClassTemplate<Int32>()
  classTemplate.func1(0)
  useGeneric(classTemplate, 0)
}
```

Class templates: importing as real generic structs

If we know the complete set of allowed type arguments to a C++ struct template, we could import it as an actual Swift generic struct. Every method of that struct will perform dynamic dispatch based on type parameters. See the section about function templates for more details.

Mapping SFINAE

We can map SFINAE to generic constraints and to conditional extensions.

TODO

Variadic templates

TODO

Non-type template parameters

TODO

Template template parameters

TODO

Overloaded operators

TODO: Mapping C++'s operators to Equatable, Comparable, Hashable.

`operator*`, `operator->`

TODO

`operator[]`

`operator[]` semantically maps well to Swift's `subscript`.

```
// C++ header.

class MyCxxContainer {
public:
  const double& operator[](int i) const;
  double& operator[](int i);
};
```

```
// C++ header imported in Swift.

struct MyCxxContainer {
  public subscript(_ i: Int) -> Double { get set }
}
```

The synthesized `subscript` uses the `_read` and `_modify` generalized accessors to forward the memory address to the caller.

```
// Implementation details of the imported API.

struct MyCxxCollection {
  // const double& operator[](int i) const;
  private func _operatorBracketsConst(_ i: Int) -> UnsafePointer<Double>

  // double& operator[](int i);
  private func _operatorBrackets(_: Int) -> UnsafeMutablePointer<Double>

  // Swifty API.
  public subscript(_ i: Int) -> Double {
    _read {
      yield _operatorBracketsConst(i).pointee
    }
    _modify {
```

```
      yield &_operatorBrackets(i).pointee
    }
  }
}
```

`operator()`

Swift has an equivalent for C++'s `operator()`: `callAsFunction` (introduced in SE-0253: Callable values of user-defined nominal types).

```
// C++ header.

class MultiplyIntDouble {
public:
  double operator()(int x, double y);
};

// C++ header imported in Swift.

struct MultiplyIntDouble {
  public func callAsFunction(_ x: Int, _ y: Double) -> Double { ... }
}
```

This mapping rule would handle the call operator of `std::function`, `function_ref` and similar types.

Literal operators

TODO

Exceptions

Background

C++ unfortunately has the opposite default to Swift where exception throwing is concerned. Any C++ function that is not explicitly marked `noexcept` must be assumed to be potentially throwing, even though many such functions don't throw in practice.

If we imported all of these functions into Swift as `throws` functions, the code calling them would be littered with `try!`s. Requiring all imported functions (and everything they call transitively) to be marked `noexcept` also seems excessively burdensome and is not idiomatic C++.

Baseline functionality: Import functions as non-throwing, terminate on uncaught C++ exceptions

In the first iteration of C++ interop, we will import all C++ functions as non-throwing Swift functions. If a C++ function called from Swift throws an

exception that is not caught within the C++ code, the program will terminate.

This approach is similar to that taken by the Python interop library, which also terminates the program by default if a Python exception is raised and not caught within the Python code.

If exceptions thrown in C++ need to be handled in Swift, this can be done by writing a C++ wrapper that catches the exception and returns a corresponding error object.

Extended functionality: Optionally propagate exceptions to Swift

If we find that developers routinely need to handle C++ exceptions in Swift, writing C++ wrappers as described above will obviously not be a satisfactory solution. In this case, we will add an option to propagate C++ exceptions as Swift exceptions; this will be backward compatible with the baseline approach described above.

We again take inspiration from Python interop. There, appending `.throwing` to the function name calls a variant of the function that propagates the Python exception to Swift.

It isn't really practical to do something exactly analogous in C++ interop. We could import each function twice and identify the throwing version by a name suffix or a dummy parameter. However, this solution would bloat the interface of imported classes, and it would not be universal: Name suffixes can't be used with constructors or operators; a dummy parameter could be used with most constructors, but still doesn't work for default constructors or operators.

Instead, we propose extending Swift with a `throws!` marker. A function marked with `throws!` can potentially throw exceptions, but the function does not need to be called with `try`. If the function *is* called with `try`, exceptions are handled as they would be for a normal throwing function. If the function is not called with `try` and the function raises an exception, the program is terminated. These semantics are close to C++ exception handling semantics.

All C++ functions that are not marked `noexcept` would be imported as `throws!` functions; `noexcept` functions would be imported as non-throwing functions.

This brief sketch obviously leaves many questions unanswered on the detailed semantics that a `throws!` feature would have, for example whether user-written Swift code should be allowed to use `throws!` – see also this forum discussion. Before we take any steps towards implementing C++ exception proagation, we will submit a formal Swift Evolution proposal for the `throws!` feature.

The other question to answer is how we would map the C++ exception to a corresponding Swift object implementing `Error`. In theory, C++ allows objects of any copy-initializable type to be thrown. In practice, most user-defined C++ exceptions derive from `std::exception`, so it would be natural to propagate C++ exceptions as the Swift-imported equivalent of `std::exception` and make

that type implement `Error`. We could add a separate fallback error type (e.g. `CxxUnknownException`) for the case where the exception does not derive from `std::exception`.

As `std::exception` is a polymorphic type, the details of how `std::exception` will be represented in Swift will need to wait until we have finalized how polymorphism is handled (see also the section on virtual member functions).

Implementation

Many common C++ ABIs allow the default behavior (terminate the program if a C++ exception is thrown) to be implemented at zero cost. In particular, it is not necessary to wrap every C++ function in a synthesized try-catch block.

For example, the Itanium C++ ABI defines a so-called personality routine that is called for each stack frame as the stack is being unwound. The idea is that different languages can have different personality routines, so that different languages can co-exist on the stack and can each define their own stack frame cleanup logic.

The stack unwinding infrastructure finds the correct personality routine by consulting a so-called unwind table, which maps program counter ranges to personality routines.

We would define unwind table entries covering all Swift code that does not attempt to handle C++ exceptions and have these entries map to a personality routine that simply terminates the program.

If exception support is turned off in the C++ compiler by passing `-Xcc -fno-exceptions` to `swiftc`, we assume that the C++ code never throws exceptions and will not emit any unwind table entries for Swift code.

Atomics

TODO

Importing non-const pointer as extra return values

TODO

Enhancing C++ API mapping into Swift with bridging

Bridging data types with different memory layout

When we import a C++ `T` into Swift as a type `U`, in the general case we must ensure that `T` and `U` have identical memory layout. For example, Swift's `Int8`

and C++'s `int8_t` have identical memory layout, and therefore, we can import `int8_t` as `Int8` everywhere.

If `T` and `U` have even slightly different memory layouts, we must invoke a bridging function at the interop boundary. Further sections in this document provide many examples of such type pairs, but to give an example, consider `std::vector<T>` in C++, which is semantically similar to `Array<T>` in Swift, but has a completely different memory layout. If we want to map them to one another, but must keep the memory layout of both types unchanged, we must convert the `std::vector<T>` value into an `Array<T>` value when the program execution crosses the language boundary.

For example:

```
// C++ header.

std::vector<int> IncrementVectorValues(std::vector<int> v);

// C++ header imported in Swift.

// Note: no C++ vectors!
func IncrementVectorValues(_ v: [CInt]) -> [CInt]

// Using the imported C++ API in Swift.

func callIncrementVectorValues() -> CInt {
  var xs: [CInt] = IncrementVectorValues([10, 20, 30])
  return xs[0] // = 11.
}
```

Bridging: behind the scenes

Here's a simplified description of how Swift implements bridging. Behind the scenes, the function `IncrementVectorValues` is imported as is, with C++ data types:

```
// C header imported in Swift, behind the scenes:

func _cxx_IncrementVectorValues(std.vector<CInt> v) -> std.vector<CInt>
```

C++ bridging support code provides a function that converts a `std::vector` to a Swift `Array` and vice versa:

```
// Swift standard library, C++ support module.

func StdVectorToArray<T>(_ vector: std.vector<T>) -> [T]
func ArrayToStdVector<T>(_ array: [T]) -> std.vector<T>
```

Each caller is transformed to call the bridging function:

```
// Using the imported C++ API in Swift: code written by the user.
```

```
func callIncrementVectorValues() -> CInt {
  var xs: [CInt] = IncrementVectorValues([10, 20, 30])
  return xs[0] // = 11.
}

// Using the imported C++ API in Swift: code rewritten by the type checker.

func callGetVector() -> CInt {
  var xs: [CInt] =
    StdVectorToArray(_cxx_IncrementVectorValues(ArrayToStdVector([10, 20, 30])))
  return xs[0] // = 11.
}
```

A naive implementation of `StdVectorToArray` and `ArrayToStdVector` would have to copy the container's elements. However, with enough cooperation between the C++ standard library and the Swift standard library, Swift `Array` could take ownership of `std::vector`'s storage and vice versa in O(1), avoiding making a copy. Such cooperation would likely require changing the ABI of at least one of the types, so it would not be feasible to implement in all environments.

Bridging does not work in every context

Bridging between types with different memory layouts does not work universally. For example, just because a C++ type `T` can be bridged to a Swift type `U`, it does not follow that we can bridge a `T*` to an `UnsafeMutablePointer<U>`.

```
// C++ header.

const std::vector<int> *GetPtrToVector();

// C++ header imported in Swift.

// We can't bridge types like in the example below, it is not implementable.
// The underlying C++ function returns a vector with unclear ownership semantics,
// but we need to return a pointer to a Swift Array.
// If the bridging code allocates the Swift Array on the heap, what would own it?
// If the bridging code does not allocate a new Array, where does it get the
// storage from?
func GetPtrToVector() -> UnsafePointer<[CInt]>

// OK.
func GetPtrToVector() -> UnsafePointer<std.vector<CInt>>
```

Therefore, we can perform non-trivial bridging only in a limited number of contexts. In all remaining places C++ types must be imported according to the general rules.

As a consequence, every non-trivial bridging solution needs a fallback that does not require adjusting the memory layout.

Bridging `std::function`

`std::function` in C++ is similar to closures in Swift. The primary difference is that `std::function` has an empty state, similarly to function pointers, while Swift closures don't.

Due to allowing an empty state, `std::function` must be mapped to an optional Swift closure, unless annotated as non-nullable.

To convert a non-empty `std::function` value to a Swift closure, we need to define a thunk and allocate a closure context on the heap. The new closure context will point to a copy of the `std::function` value. The thunk will use a Swift calling convention, and it will forward the arguments to `std::function::operator()()` using the C++ calling convention.

A thunk like this will not cost much in terms of code size. The primary cost of the thunk will come from dynamic dispatch. We will have double dynamic dispatch: first one at the callsite that calls the Swift closure and ends up calling the thunk, and second one from the thunk to the C++ code wrapped in `std::function`. Another issue is branch predictor pollution: there will be a single thunk backing all closures with the same signature. Indirect branches from thunk will be difficult to predict for the branch predictor.

There is additional cost contributed by reference counting: closures in Swift are reference counted, while `std::function` isn't.

Finally, allocating a new closure context on the heap is a cost to be paid on every conversion from a C++ `std::function` to a Swift closure.

Bridging `std::string`

Ergonomically, the best mapping of C++'s `std::string` is Swift's `String`. Both types support nul bytes (the conversion has to take them into account and cannot assume nul-terminated strings). However, there's a semantic gap: Swift strings require text to be valid UTF-8.

We have a couple of options to address the UTF-8 issue:

- Trust C++ code to only store UTF-8 text in strings, and map them to Swift's `String` where possible. Perform a runtime check for UTF-8 validity during the conversion. Allow to annotate `std::string`s that store binary data, and map those as either Swift's `Array` or `std::string`.
- Assume `std::string`s store binary data, and map them to according to the regular interop rules to `cxx_std.string`. Allow users to annotate UTF-8 strings as such, and map them to Swift's `String`.

The annotations could be inferred with a dynamic analysis tool that would track which `std::string`s contain binary data.

Here's an example of API mapping if we decide to trust `std::string`s to store UTF-8 data.

```
// C++ header.

class Employee {
public:
  std::string DebugDescription() const;
  [[swift::import_as_std_string]] std::string SerializedAsProtobuf() const;
};

// C++ header imported in Swift.

struct Employee {
  func DebugDescription() -> String
  func SerializedAsProtobuf() -> std.string
}
```

To avoid copying data unnecessarily, we need the C++ and Swift standard libraries to cooperate, so that Swift's `String` can take ownership of `std::string`'s heap allocation, and vice versa.

Character data in `Swift.String` is stored in a reference-counted object. `Swift.String` can take ownership of `std::string` by transferring ownership of `std::string` to a newly-allocated reference-counted object. Swift/Objective-C bridging of `NSString` already works exactly this way.

On non-Objective-C platforms, `Swift.String` assumes that the character data is tail-allocated in the reference-counted object; this assumption will have to be replaced with a branch. Therefore, adding this sort of bridging on non-Objective-C platforms will create a certain (small) performance penalty. It is important to note that this performance penalty has been always affecting Objective-C platforms and it is considered acceptable.

We could eliminate the branch on non-Objective-C platforms by changing `Swift.String` there to always use a reference-counted object with an `std::string` in it as the backing storage. The disadvantage of such memory layout is that `Swift.String` would need to dereference two levels of indirection to access to character data, instead of one like we have today.

`std::string` can't take ownership of `Swift.String`'s reference-counted object, because `std::string` has no branches in its code that retrieves the pointer to the character data and in its deallocation code. Adding branches there is certainly possible, but will likely lead to regressions for many C++ applications, including ones that don't use Swift. Therefore, this approach looks like a non-starter.

Therefore, the only viable approach for O(1) bidirectional bridging is having

`Swift.String` use `std::string` as its backing storage, and taking the penalty of a double-indirection to access character data.

If one-way C++-to-Swift bridging is sufficient, it can be implemented with a branch in the `Swift.String` code that retrieves a pointer to character data.

We can implement one-way O(1) Swift-to-C++ bridging by embedding a fake `std::string` in the `Swift.String` buffer. Such a `std::string` instance would not be full-fledged, and would be only good for reading from it; its destructor can never be called.

Bridging `std::vector`

Ergonomically, the best mapping of C++'s `std::vector` is Swift's `Array`.

Except for the case of `std::vector<bool>`, the constraints for `Swift.Array` taking ownership of `std::vector`'s element buffer and vice versa are identical to string bridging.

Bridging `std::set`, `std::map`

Swift standard library does not provide collections with semantics similar to `std::set` and `std::map`, so they can't be meaningfully bridged to any Swift vocabulary type. Therefore, they will be imported as `std.set` and `std.map`.

It would be useful to provide ergonomic ways to convert `std.set` and `std.map` to `Swift.Set` and `Swift.Dictionary`, since `std::set` and `std::map` are sometimes used in C++ APIs (especially older ones) when the element order doesn't matter.

Bridging `std::unordered_set`, `std::unordered_map`

The most ergonomic mapping of `std::unordered_set` and `std::unordered_map` are `Swift.Set` and `Swift.Dictionary`.

The constraints for these Swift types taking ownership of C++'s storage buffers and vice versa are quite similar to string bridging. An interesting new point to highlight is that the `Swift.Set` and `Swift.Dictionary` types would need to use the hash function defined in C++ for C++ types. It should be possible to expose the C++ hash function in Swift as a conformance to the `Hashable` protocol, or as a conformance to a new C++-specific protocol for hashing.

Bridging `std::tuple`, `std::pair`

Ergonomically, the best mapping of C++'s `std::tuple` and `std::pair` are Swift's tuples.

Tuple and pair elements have to be copied to implement non-trivial bridging. It is unreasonable to introduce a dual representation and require a branch to

access tuple elements either in C++ or Swift.

However, there are only so many different ways to lay out a `std::pair` in memory. In fact, it is almost certainly a struct with exactly two data members, `first` and `second`, in that order. When `std::pair<T, U>` and Swift's tuple `(T, U)` happen to use the same layout, we could bridge them trivially, without copying the data.

```
// C++ header.

std::pair<int, int> *GetPairPtr();

// C++ header imported in Swift.

UnsafeMutablePointer<(Int, Int)> GetPairPtr()
```

`std::tuple` is more complex and has more viable implementation options compared to `std::pair`. For example, `std::tuple` is laid out in forward order by libc++, and in backward order by libstdc++. Therefore, if libc++ is used, there's a good chance that the memory layout of `std::tuple` matches the memory layout of the corresponding Swift tuple, and we could bridge them trivially.

The disadvantage of the techniques described above is that they depend on implementation details of the standard library, and as those change (or if the user switches to a different standard library altogether), the API in Swift might change as well.

Bridging `std::span`

The difficulty with `std::span<T>` is that it has a different API in Swift depending on whether `T` is `const` or not.

```
// C++ standard library.

template<typename T>
class span {
public:
  constexpr T& front() const noexcept;
};

// C++ standard library imported in Swift.

// When T is non-const.
struct span<T> {
  public func front() -> UnsafeMutablePointer<T>
}

// When T is const.
```

```
struct span<T> {
  public func front() -> UnsafePointer<T>
}

// Need to choose one way to import, can't have both!
```

The easiest way to deal with this is to only import `std::span<T>` as full specialization (the fallback way to import templates). However, that would not be very ergonomic, and would prevent other templates that use `std::span` from being imported as generics in Swift.

A more ergonomic way would be to import `std::span` as two generic structs in Swift, selecting the appropriate one depending on constness.

```
// C++ standard library imported in Swift.

struct mutable_span<T> {
  public func front() -> UnsafeMutablePointer<T>
}

struct const_span<T> {
  public func front() -> UnsafePointer<T>
}

// C++ header.

std::span<int> GetMutableIntSpan();
std::span<const int> GetConstIntSpan();

// C++ header imported in Swift.

func GetMutableIntSpan() -> std.mutable_span<Int>
func GetConstIntSpan() -> std.const_span<Int>
```

Standard C++ containers in general

It seems appropriate to provide an overlay that adds `Swift.Collection` conformances to C++ container types.

Custom C++ containers

The importer could try to recognize custom C++ containers and synthesize `Swift.Collection` conformances, however, that seems fragile: custom containers often don't implement the complete container API as required by the standard, or deviate from it in subtle ways.

It seems best to add `Swift.Collection` conformances in overlays.

Bridging `absl::flat_hash_{set,map}`, `absl::node_hash_{set,map}`

Projects that use Abseil often use Abseil hash tables a lot more than standard containers. However the same is also true about other libraries, like Boost or Qt. Therefore, privileging a third-party library in C++/Swift interop seems inappropriate. Abseil, Boost, and Qt should be treated just like user-written code. Library vendors can provide annotations and Swift overlays to improve the API in Swift.

Enhancing C++ API mapping into Swift with annotations

Annotation workflow

Annotations in C headers are employed by the Swift / C interop today (for example, nullability annotations). However, these annotations must be added to headers manually. How could tools infer annotations?

Online inference at compile-time. The Swift compiler could infer annotations from information available in the headers. For example:

```
// example.h

class Example {
public:
  int *get_mutable_value() { return &value; }
  // The Swift compiler infers that the return type is `int * _Nonnull` because
  // the function never returns nullptr.
};
```

The problem with this approach is that it is making API decisions based on API implementation details, without letting the API owner to intervene. It is going to be fragile in the long run: no-op changes to implementation details of C++ APIs can tip annotation inference one way or the other, and unintentionally change the Swift API. Imagine that the example above is refactored to:

```
// example.h

class Example {
public:
  int *get_mutable_value() { return get_mutable_value_impl(); }
private:
  int *get_mutable_value_impl(); // defined in example.cc
};

// The Swift compiler does not know whether `get_mutable_value_impl()`
// can return nullptr or not, therefore its return type is
```

```
// `int * _Null_unspecified`. Therefore, the return type of
// `get_mutable_value()` is now changed to `int * _Null_unspecified` as well.
```

API declarations should be the source of truth for API information; this also aligns with expectations of C++ API owners.

Offline inference. The concerns about inspecting implementation details for inference would be alleviated if inference tooling produced a source code patch that would be reviewed by a human and adjusted if necessary before being committed.

The ideal option is to add annotations directly to the header, to make the API contract obvious to all readers and API owners. However, sometimes API owners don't want to change the header, or can't be reached. In those cases, annotations can be added to a sidecar file, removing the need to change the header.

A sidecar annotation file allows to add arbitrary attributes to declarations parsed from a header file. You can find examples of such files in the apinotes directory. APINotes files are handled by the APINotes library in Clang. Clang reads an APINotes file alongside the header file; Clang injects attributes specified by APINotes into the AST parsed from the header.

Tooling to infer nullability annotations

Nullability annotations in C++ headers are useful not only for Swift / C++ interop; they also document the intended API contract to C++ API users. Nullability annotations can be validated by static and dynamic analysis tooling in C++ (e.g., UBSan can be extended to detect assignments of `nullptr` to `_Nonnull` pointers).

To infer nullability annotations we can use a combination of static and dynamic analysis. Static analysis could infer nullability in more obvious cases – along with some confidence score, while dynamic analysis could provide insight into arbitrarily complex APIs and data structures, with the usual caveat that the inference results are only as good as the test cases.

There is a lot of existing research about detecting nullability issues in C and C++ with static analysis, so we are not discussing the design of potential static analysis tooling here. However, we are not aware of existing dynamic analysis tooling that infers nullability.

Nullability inference based on dynamic analysis will use compile-time instrumentation that is similar to UBSan for references, but instead of producing a fatal error upon a nullptr store, we will record this fact and continue. For each source-level pointer variable and class member declaration, the compiler will allocate a global boolean flag that tracks nullability, and a usage counter, which will be used to gauge confidence. The nullability flags are initialized to "false" at program startup, usage counters are initialized to zero. Each store into the pointer variables is instrumented, and when a nullptr is stored, the corresponding

flag is set to "true". Either way, the usage counter is incremented. At program shutdown (or when signalled somehow), the inferred nullability flags and usage counters are saved to disk.

APIs that take a pointer and count

Many C and C++ APIs take a pointer and count. It would be nice if we could import them as a buffer pointer in Swift, given appropriate annotations:

```
// C++ header.

void example(int *xs, int count);
// TODO: figure out what annotations might look like.

// C++ header imported in Swift.

func example(_ xs: UnsafeMutableBufferPointer<CInt>)
```

Current state of art: importing Swift code into C

The Swift compiler generates a header that C code can use.

TODO: add details.

Importing Swift APIs into C++

TODO

Resilient types

The semantic gap here is related to resilient types – their size is unknown at compile time. `std::vector<T>` needs to know `sizeof(T)` at compile time. This is a common problem in Swift-to-C++ bridging in general, and it would be solved with boxing. Boxes for resilient Swift types would have a fixed size.

Forum discussions

The topic of Swift/C++ interoperability has been discussed on Swift forums before:

- C++ Interop
- C++ / Objective-C++ Interop
- C++ support in Swift