How Swift imports C APIs

When Swift imports a module, or parses a bridging header from a C-based language (C, Objective-C), the APIs are mapped into Swift APIs and can be used directly from Swift code. This provides the basis for Swift's Foreign Function Interface (FFI), providing interoperability with existing libraries written in C-based languages.

This document describes how APIs from C-based languages are mapped into Swift APIs. It is written for a broad audience, including Swift and C users who might not be language experts. Therefore, it explains some advanced concepts where necessary.

- Names, identifiers and keywords
  - Unicode
  - Names that are keywords in Swift
  - Name translation
  - Name customization
- Size, stride, and alignment of types
- Fundamental types
- Free functions
  - Argument labels
  - Variadic arguments
  - Inline functions
- Global variables
- Pointers to data
- Nullable and non-nullable pointers
- Incomplete types and pointers to them
- Function pointers
- Fixed-size arrays
- Structs
- Unions
- Enums
- Typedefs
- Macros

Names, identifiers and keywords

Unicode

C (and C++) permit non-ASCII Unicode code points in identifiers. While Swift does not permit arbitrary Unicode code points in identifiers (so compatibility might not be perfect), it tries to allow reasonable ones. However, C, Objective-C, and C++ code in practice does not tend to use non-ASCII code points in identifiers, so mapping them to Swift is not an important concern.

Names that are keywords in Swift

Some C and C++ identifiers are keywords in Swift. Despite that, such names are imported as-is into Swift, because Swift permits escaping keywords to use them as identifiers:

```
// C header.

// The name of this function is a keyword in Swift.
void func();
```

```
// C header imported in Swift.

// The name of the function is still `func`, but it is escaped to make the
// keyword into an identifier.
func `func`()
```

```
// Swift user.

func test() {
  // Call the C function declared above.
  `func`()
}
```

Name translation

Names of some C declarations appear in Swift differently. In particular, names of enumerators (enum constants) go through a translation process that is hardcoded into the compiler. For more details, see Name Translation from C to Swift.

Name customization

As a general principle of Swift/C interoperability, the C API vendor has broad control over how their APIs appear in Swift. In particular, the vendor can use the `swift_name` Clang attribute to customize the names of their C APIs in order to be more idiomatic in Swift. For more details, see Name Translation from C to Swift.

Size, stride, and alignment of types

In C, every type has a size (computed with the `sizeof` operator), and an alignment (computed with `alignof`).

In Swift, types have size, stride, and alignment.

The concept of alignment in C and Swift is exactly the same.

Size and stride is more complicated. In Swift, stride is the distance between two elements in an array. The size of a type in Swift is the stride minus the tail padding. For example:

```
struct SwiftStructWithPadding {
  var x: Int16
  var y: Int8
}

print(MemoryLayout<SwiftStructWithPadding>.size) // 3
print(MemoryLayout<SwiftStructWithPadding>.stride) // 4
```

C's concept of size corresponds to Swift's stride (not size!) C does not have an equivalent of Swift's size.

Swift tracks the exact size of the data stored in a type so that it can pack additional data into bytes that otherwise would be wasted as padding. Swift also tracks possible and impossible bit patterns for each type, and reuses impossible bit patterns to encode more information, similarly to `llvm::PointerIntPair` and `llvm::PointerUnion`. The language does this automatically, and transparently for users. For example:

```
// This enum takes 1 byte in memory, which has 256 possible bit patterns.
// However, only 2 bit patterns are used.
enum Foo {
  case A
  case B
}

print(MemoryLayout<Foo>.size) // 1
print(MemoryLayout<Foo?>.size) // also 1: `nil` is represented as one of the 254 bit patter
```

Nevertheless, for types imported from C, the size and the stride are equal.

```
// C header.

struct CStructWithPadding {
  int16_t x;
  int8_t y;
};
```

```
// C header imported in Swift.

struct CStructWithPadding {
  var x: Int16
  var y: Int8
}
```

```
print(MemoryLayout<CStructWithPadding>.size) // 4
print(MemoryLayout<CStructWithPadding>.stride) // 4
```

Fundamental types

In C, certain types (`char`, `int`, `float` etc.) are built into the language and into the compiler. These builtin types have behaviors that are not possible to imitate in user-defined types (e.g., usual arithmetic conversions).

C and C++ standard libraries provide headers that define character and integer types that are typedefs to one of the underlying builtin types, for example, `int16_t`, `size_t`, and `ptrdiff_t`.

Swift does not have such builtin types. Swift's equivalents to C's fundamental types are defined in the Swift standard library as ordinary structs. They are implemented using compiler intrinsics, but the API surface is defined in ordinary Swift code:

```
// From the Swift standard library:

struct Int32 {
  internal var _value: Builtin.Int32

  // Note: `Builtin.Xyz` types are only accessible to the standard library.
}

func +(lhs: Int32, rhs: Int32) -> Int32 {
  return Int32(_value: Builtin.add_Int32(lhs._value, rhs._value))
}
```

Memory layout of these "fundamental" Swift types does not have any surprises, hidden vtable pointers, metadata, or reference counting; it is exactly what you expect from a corresponding C type: just the data, stored inline. For example, Swift's `Int32` is a contiguous chunk of four bytes, all of which store the number.

Fundamental types in C, with a few exceptions, have an implementation-defined size, alignment, and stride (distance between two array elements).

Swift's integer and floating point types have fixed size, alignment and stride across platforms, with two exceptions: `Int` and `UInt`. The sizes of `Int` and `UInt` match the size of the pointer on the platform, similarly to how `size_t`, `ptrdiff_t`, `intptr_t`, and `uintptr_t` have the same size as a pointer in most C implementations. `Int` and `UInt` are distinct types, they are not typealiases to explicitly-sized types. An explicit conversion is required to convert between `Int`, `UInt`, and any other integer type, even if sizes happen to match on the current platform.

The table below summarizes mapping between fundamental types of C and C++ and Swift types. This table is based on `swift.git/include/swift/ClangImporter/BuiltinMappedTypes.def`

C and C++ types	Swift types
C `_Bool`, C++ `bool`	`typealias CBool = Bool`
`char`, regardless if the target defines it as signed or unsigned	`typealias CChar = Int8`
`signed char`, explicitly signed	`typealias CSignedChar = Int8`
`unsigned char`, explicitly unsigned	`typealias CUnsignedChar = UInt8`
`short`, `signed short`	`typealias CShort = Int16`
`unsigned short`	`typealias CUnsignedShort = UInt16`
`int`, `signed int`	`typealias CInt = Int32`
`unsigned int`	`typealias CUnsignedInt = UInt32`
`long`, `signed long`	Windows x86_64: `typealias CLong = Int32`Everywhere else: `typealias CLong = Int`
`unsigned long`	Windows x86_64: `typealias CUnsignedLong = UInt32`Everywhere else: `typealias CUnsignedLong = UInt`
`long long`, `signed long long`	`typealias CLongLong = Int64`
`unsigned long long`	`typealias CUnsignedLongLong = UInt64`

C and C++ types	Swift types
`wchar_t`, regardless if the target defines it as signed or unsigned	`typealias CWideChar = Unicode.ScalarUnicode.Scalar` is a wrapper around `UInt32`
`char8_t` (proposed for C++20)	Not mapped
`char16_t`	`typealias CChar16 = UInt16`
`char32_t`	`typealias CChar32 = Unicode.Scalar`
`float`	`typealias CFloat = Float`
`double`	`typealias CDouble = Double`
`long double`	`CLongDouble`, which is a typealias to `Float80` or `Double`, depending on the platform.There is no support for 128-bit floating point.

First of all, notice that C types are mapped to Swift typealiases, not directly to the underlying Swift types. The names of the typealiases are based on the original C types. These typealiases allow developers to easily write Swift code and APIs that work with APIs and data imported from C without a lot of `#if` conditions.

This table is generally unsurprising: C types are mapped to corresponding explicitly-sized Swift types, except for the C `long`, which is mapped to `Int` on 32-bit and 64-bit LP64 platforms. This was done to enhance portability between 32-bit and 64-bit code.

The difficulty with the C `long` type is that it can be 32-bit or 64-bit depending on the platform. If C `long` was mapped to an explicitly-sized Swift type, it would map to different Swift types on different platforms, making it more difficult to write portable Swift code (the user would have to compile Swift code for both platforms to see all errors; fixing compilation errors on one platform can break another platform.) By mapping `long` a distinct type, `Int`, the language forces the user to think about both cases when compiling for either platform.

Nevertheless, mapping C `long` to `Int` does not work universally. Specifically, it does not work on LLP64 platforms (for example, Windows x86_64), where C `long` is 32-bit and Swift's `Int` is 64-bit. On LLP64 platforms, C `long` is mapped to Swift's explicitly-sized `Int32`.

Typedefs for integer types in the C standard library are mapped like this (from `swift.git/swift/lib/ClangImporter/MappedTypes.def`:

C and C++ types	Swift types
`uint8_t`	`UInt8`
`uint16_t`	`UInt16`
`uint32_t`	`UInt32`
`uint64_t`	`UInt64`
`int8_t`	`Int8`
`int16_t`	`Int16`
`int32_t`	`Int32`
`int64_t`	`Int64`
`intptr_t`	`Int`
`uintptr_t`	`UInt`
`ptrdiff_t`	`Int`
`size_t`	`Int`
`rsize_t`	`Int`
`ssize_t`	`Int`

```
// C header.

double Add(int x, long y);
```

```
// C header imported in Swift.

func Add(_ x: CInt, _ y: CLong) -> CDouble
```

Free functions

C functions are imported as free functions in Swift. Each type in the signature of the C function is mapped to the corresponding Swift type.

Argument labels

Imported C functions don't have argument labels in Swift by default. Argument labels can be added by API owners through annotations in the C header.

```
// C header.
```

```
#define SWIFT_NAME(X) __attribute__((swift_name(#X)))

// No argument labels by default.
void drawString(const char *, int xPos, int yPos);

// The attribute specifies the argument labels.
void drawStringRenamed(const char *, int xPos, int yPos)
    SWIFT_NAME(drawStringRenamed(_:x:y:));
```

```
// C header imported in Swift.

func drawString(_: UnsafePointer<CChar>!, _ xPos: CInt, _ yPos: CInt)
func drawStringRenamed(_: UnsafePointer<CChar>!, x: CInt, y: CInt)

drawString("hello", 10, 20)
drawStringRenamed("hello", x: 10, y: 20)
```

Variadic arguments

C functions with variadic arguments are not imported into Swift, however, there are no technical reasons why they can't be imported.

Note that functions with `va_list` arguments are imported into Swift. `va_list` corresponds to `CVaListPointer` in Swift.

C APIs don't define a lot of variadic functions, so this limitation has not caused a big problem so far.

Often, for each variadic function there is a corresponding function that takes a `va_list` which can be called from Swift. A motivated developer can write an overlay that exposes a Swift variadic function that looks just like the C variadic function, and implement it in terms of the `va_list`-based C API. You can find examples of such overlays and wrappers by searching for usages of the `withVaList` function in the `swift.git/stdlib` directory.

See also Apple's documentation about this topic: Use a CVaListPointer to Call Variadic Functions.

Inline functions

Inline C functions that are defined in headers are imported as regular Swift functions. However, unlike free functions, inline functions require the caller to emit a definition of the function, because no other translation unit is guaranteed to provide a definition.

Therefore, the Swift compiler uses Clang's CodeGen library to emit LLVM IR for the C inline function. LLVM IR for C inline functions and LLVM IR for Swift code is put into one LLVM module, allowing all LLVM optimizations (like inlining) to work transparently across language boundaries.

Global variables

Global C variables are imported as Swift variables or constants, depending on constness.

```
// C header.

extern int NumAlpacas;
extern const int NumLlamas;
```

```
// C header imported in Swift.

var NumAlpacas: CInt
let NumLlamas: CInt
```

Pointers to data

C has one way to form a pointer to a value of type `T` – `T*`.

Swift language does not provide a builtin pointer type. The standard library defines multiple pointer types:

- `UnsafePointer<T>`: equivalent to `const T*` in C.
- `UnsafeMutablePointer<T>`: equivalent to `T*` in C, where `T` is non-const.
- `UnsafeRawPointer`: a pointer for accessing data that is not statically typed, similar to `const void*` in C. Unlike C pointer types, `UnsafeRawPointer` allows type punning, and provides special APIs to do it correctly and safely.
- `UnsafeMutableRawPointer`: like `UnsafeRawPointer`, but can mutate the data it points to.
- `OpaquePointer`: a pointer to typed data, however the type of the pointee is not known, for example, it is determined by the value of some other variable, or the type of the pointee is not representable in Swift.
- `AutoreleasingUnsafeMutablePointer<T>`: only used for Objective-C interoperability; corresponds to an Objective-C pointer T `__autoreleasing *`, where `T` is an Objective-C pointer type.

C pointer types can be trivially imported in Swift as long as the memory layout of the pointee is identical in C and Swift. So far, this document only described primitive types, whose memory layout in C and Swift is indeed identical, for example, `char` in C and `Int8` in Swift. Pointers to such types are trivial to import into Swift, for example, `char*` in C corresponds to `UnsafeMutablePointer<Int8>!` in Swift.

```
// C header.

void AddSecondToFirst(int *x, const long *y);
```

```
// C header imported in Swift.

func AddSecondToFirst(_ x: UnsafeMutablePointer<CInt>!, _ y: UnsafePointer<CLong>!)
```

Nullable and non-nullable pointers

Any C pointer can be null. However, in practice, many pointers are never null. Therefore, code often does not expect certain pointers to be null and does not handle null values gracefully.

C does not provide a way to distinguish nullable and non-nullable pointers. However, Swift makes this distinction: all pointer types (for example, `UnsafePointer<T>` and `UnsafeMutablePointer<T>`) are non-nullable. Swift represents the possibility of a missing value with a type called "Optional", spelled `T?` in the shorthand form, or `Optional<T>` fully. The missing value is called "nil". For example, `UnsafePointer<T>?` (shorthand for `Optional<UnsafePointer<T>>`) can store a nil value.

Swift also provides a different syntax for declaring an optional, `T!`, which creates a so-called "implicitly unwrapped optional". These optionals are automatically checked for nil and unwrapped if it is necessary for the expression to compile. Unwrapping a `T?` or a `T!` optional that contains nil is a fatal error (the program is terminated with an error message).

Formally, since any C pointer can be null, C pointers must be imported as optional unsafe pointers in Swift. However, that is not idiomatic in Swift: optional should be used when value can be truly missing, and when it is meaningful for the API. C APIs do not provide this information in a machine-readable form. Information about which pointers are nullable is typically provided in free-form documentation for C APIs, if it is provided at all.

Clang implements an extension to the C language that allows C API vendors to annotate pointers as nullable or non-nullable.

Quoting the Clang manual:

> The `_Nonnull` nullability qualifier indicates that null is not a meaningful value for a value of the `_Nonnull` pointer type. For example, given a declaration such as:

```
int fetch(int * _Nonnull ptr);
```

> a caller of `fetch` should not provide a null value, and the compiler will produce a warning if it sees a literal null value passed to fetch. Note that, unlike the declaration attribute `nonnull`, the presence of `_Nonnull` does not imply that passing null is undefined behavior: `fetch` is free to consider null undefined behavior or (perhaps for backward-compatibility reasons) defensively handle null.

`_Nonnull` C pointers are imported in Swift as non-optional `UnsafePointer<T>` or `UnsafeMutablePointer<T>`, depending on the constness of the pointer.

```
// C declaration above imported in Swift.

func fetch(_ ptr: UnsafeMutablePointer<CInt>) -> CInt
```

Quoting the Clang manual:

> The `_Nullable` nullability qualifier indicates that a value of the `_Nullable` pointer type can be null. For example, given:

```
int fetch_or_zero(int * _Nullable ptr);
```

> a caller of `fetch_or_zero` can provide null.

`_Nullable` pointers are imported in Swift as `UnsafePointer<T>?` or `UnsafeMutablePointer<T>?`, depending on the constness of the pointer.

```
// C declaration above imported in Swift.

func fetch_or_zero(_ ptr: UnsafeMutablePointer<CInt>?) -> CInt
```

Quoting the Clang manual:

> The `_Null_unspecified` nullability qualifier indicates that neither the `_Nonnull` nor `_Nullable` qualifiers make sense for a particular pointer type. It is used primarily to indicate that the role of null with specific pointers in a nullability-annotated header is unclear, e.g., due to overly-complex implementations or historical factors with a long-lived API.

`_Null_unspecified` and not annotated C pointers are imported in Swift as implicitly-unwrapped optional pointers, `UnsafePointer<T>!` or `UnsafeMutablePointer<T>!`. This strategy provides ergonomics equivalent to the original C API (no need to explicitly unwrap), and safety expected by Swift code (a dynamic check for null during implicit unwrapping).

These qualifiers do not affect program semantics in C and C++, allowing C API vendors to safely add them to headers for the benefit of Swift users without disturbing existing C and C++ users.

In C APIs most pointers are non-nullable. To reduce the annotation burden, Clang provides a way to mass-annotate pointers as non-nullable, and then mark exceptions with `_Nullable`.

```
// C header.

void Func1(int * _Nonnull x, int * _Nonnull y, int * _Nullable z);

#pragma clang assume_nonnull begin
```

```c
void Func2(int *x, int *y, int * _Nullable z);

#pragma clang assume_nonnull end
```

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

// Note that `Func1` and `Func2` are imported identically, but `Func2` required
// fewer annotations in the C header.

func Func1(
  _ x: UnsafeMutablePointer<CInt>,
  _ y: UnsafeMutablePointer<CInt>,
  _ z: UnsafeMutablePointer<CInt>?
)

func Func2(
  _ x: UnsafeMutablePointer<CInt>,
  _ y: UnsafeMutablePointer<CInt>,
  _ z: UnsafeMutablePointer<CInt>?
)
```

API owners that adopt nullability qualifiers usually wrap all declarations in the header with a single `assume_nonnull begin/end` pair of pragmas, and then annotate nullable pointers.

See also Apple's documentation about this topic: Designating Nullability in Objective-C APIs.

Incomplete types and pointers to them

C and C++ have a notion of incomplete types; Swift does not have anything similar. Incomplete C types are not imported in Swift in any form.

Sometimes types are incomplete only accidentally, for example, when a file just happens to forward declare a type instead of including a header with a complete definition, although it could include that header. In cases like that, to enable Swift to import the C API, it is recommended to change C headers and to replace forward declarations with `#include`s of the header that defines the type.

Incomplete types are often used intentionally to define opaque types. This is done too often, and Swift could not ignore this use case. Swift imports pointers to incomplete types as `OpaquePointer`. For example:

```c
// C header.

struct Foo;
void Print(const Foo* foo);
```

```swift
// C header imported in Swift.
// Swift can't import the incomplete type `Foo` and has to drop some type
// information when importing a pointer to `Foo`.
func Print(_ foo: OpaquePointer)
```

Function pointers

C supports only one form of function pointers: `Result (*)(Arg1, Arg2, Arg3)`.

Swift's closest native equivalent to a function pointer is a closure: `(Arg1, Arg2, Arg3) -> Result`. Swift closures don't have the same memory layout as C function pointers: closures in Swift consist of two pointers, a pointer to the code and a pointer to the captured data (the context). A C function pointer can be converted to a Swift closure; however, bridging is required to adjust the memory layout.

As discussed above, there are cases where bridging that adjusts memory layout is not possible, for example, when importing pointers to function pointers. For example, while C's `int (*)(char)` can be imported as `(CChar) -> CInt` (requires an adjustment of memory layout), C's `int (**)(char)` can't be imported as `UnsafePointer<(CChar) -> CInt>`, because the pointee must have identical memory layout in C and in Swift.

Therefore, we need a Swift type that has a memory layout identical to C function pointers, at least for such fallback cases. This type is spelled `@convention(c) (Arg1, Arg2, Arg3) -> Result`.

Even though it is possible to import C function pointers as Swift closure with a context pointer in some cases, C function pointers are always imported as `@convention(c)` "closures" (in quotes because they don't have a context pointer, so they are not real closures). Swift provides an implicit conversion from `@convention(c)` closures to Swift closures with a context.

Importing C function pointers also takes pointer nullability into account: a nullable C function pointer is imported as optional.

```c
// C header.

void qsort(
  void *base,
  size_t nmemb,
  size_t size,
  int (*compar)(const void *, const void *));

void qsort_annotated(
  void * _Nonnull base,
  size_t nmemb,
  size_t size,
  int (* _Nonnull compar)(const void * _Nonnull, const void * _Nonnull));
```

```
// C header imported in Swift.

func qsort(
  _ base: UnsafeMutableRawPointer!,
  _ nmemb: Int,
  _ size: Int,
  _ compar: (@convention(c) (UnsafeRawPointer?, UnsafeRawPointer?) -> CInt)!
)

func qsort_annotated(
  _ base: UnsafeMutableRawPointer,
  _ nmemb: Int,
  _ size: Int,
  _ compar: @convention(c) (UnsafeRawPointer, UnsafeRawPointer) -> CInt
)
```

See also Apple's documentation about this topic: Using Imported C Functions in Swift

Fixed-size arrays

C's fixed-size arrays are imported as Swift tuples.

```
// C header.

extern int x[4];
```

```
// C header imported in Swift.

var x: (CInt, CInt, CInt, CInt) { get set }
```

This mapping strategy is widely recognized as being far from optimal because ergonomics of Swift tuples does not match C's fixed-size arrays. For example, Swift tuples cannot be accessed through an index that only becomes known at runtime. If you need to access a tuple element by index, you have to get an unsafe pointer to values in a homogeneous tuple with `withUnsafeMutablePointer(&myTuple) { ... }`, and then perform pointer arithmetic.

Fixed-size arrays are a commonly requested feature in Swift, and a good proposal is likely to be accepted. Once Swift has fixed-size arrays natively in the language, we can use them to improve C interoperability.

Structs

C structs are imported as Swift structs, their fields are mapped to stored Swift properties. Bitfields are mapped to computed Swift properties. Swift structs also get a synthesized default initializer (that sets all properties to zero), and an elementwise initializer (that sets all properties to the provided values).

```
// C header.

struct Point {
  int x;
  int y;
};

struct Line {
  struct Point start;
  struct Point end;
  unsigned int brush : 4;
  unsigned int stroke : 3;
};
```

```
// C header imported in Swift.

struct Point {
  var x: CInt { get set }
  var y: CInt { get set }
  init()
  init(x: CInt, y: CInt)
}

struct Line {
  var start: Point { get set }
  var end: Point { get set }
  var brush: CUnsignedInt { get set }
  var stroke: CUnsignedInt { get set }

  // Default initializer that sets all properties to zero.
  init()

  // Elementwise initializer.
  init(start: Point, end: Point, brush: CUnsignedInt, stroke: CUnsignedInt)
}
```

Swift can also import unnamed and anonymous structs.

```
// C header.

struct StructWithAnonymousStructs {
  struct {
    int x;
  };
  struct {
    int y;
  } containerForY;
};

// C header imported in Swift.
struct StructWithAnonymousStructs {
  struct __Unnamed_struct___Anonymous_field0 {
    var x: CInt
    init()
    init(x: CInt)
  }
  struct __Unnamed_struct_containerForY {
    var y: CInt
    init()
    init(y: CInt)
  }
  var __Anonymous_field0: StructWithAnonymousStructs.__Unnamed_struct___Anonymous_field0
  var x: CInt
  var containerForY: StructWithAnonymousStructs.__Unnamed_struct_containerForY

  // Default initializer that sets all properties to zero.
  init()

  // Elementwise initializer.
  init(
    _ __Anonymous_field0: StructWithAnonymousStructs.__Unnamed_struct___Anonymous_field0,
    containerForY: StructWithAnonymousStructs.__Unnamed_struct_containerForY
  )
}
```

See also Apple's documentation about this topic: Using Imported C Structs and Unions in Swift.

Unions

Swift does not have a direct equivalent to a C union. C unions are mapped to Swift structs with computed properties that read from/write to the same underlying storage.

```
// C header.

union IntOrFloat {
  int i;
  float f;
};

// C header imported in Swift.
```

```
struct IntOrFloat {
  var i: CInt { get set } // Computed property.
  var f: CFloat { get set } // Computed property.
  init(i: CInt)
  init(f: CFloat)
  init()
}
```

See also Apple's documentation about this topic: Using Imported C Structs and Unions in Swift.

Enums

We would have liked to map C enums to Swift enums, like this:

```
// C header.

// Enum that is not explicitly marked as either open or closed.
enum HomeworkExcuse {
  EatenByPet,
  ForgotAtHome,
  ThoughtItWasDueNextWeek,
};
```

```
// C header imported in Swift: aspiration, not an actual mapping!

enum HomeworkExcuse: CUnsignedInt {
  case EatenByPet
  case ForgotAtHome
  case ThoughtItWasDueNextWeek
}
```

However, in practice, plain C enums are mapped to Swift structs like this:

```
// C header imported in Swift: actual mapping.

struct HomeworkExcuse: Equatable, RawRepresentable {
  init(_ rawValue: CUnsignedInt)
  init(rawValue: CUnsignedInt)
  var rawValue: CUnsignedInt { get }
  typealias RawValue = CUnsignedInt
}
var EatenByPet: HomeworkExcuse { get }
var ForgotAtHome: HomeworkExcuse { get }
var ThoughtItWasDueNextWeek: HomeworkExcuse { get }
```

To explain why this mapping was chosen, we need to discuss certain features of C enums:

- In C, adding new enumerators is not source-breaking; in Itanium C ABI, it is not ABI breaking either as long as the size of the enum does not change. Therefore, C library vendors add enumerators without expecting downstream breakage.
- In C, it is common to use enums as bitfields: enumerators are assigned values that are powers of two, and enum values are bitwise-or combinations of enumerators.
- Some C API vendors define two sets of enumerators: "public" enumerators that are listed in the header file, and "private" enumerators that are used only in the implementation of the library.

Due to these coding patterns, at runtime, C enums can carry values that were not listed in the enum declaration at compile time (either such values were added after the code was compiled, or they are a result of an intentional cast from an integer to an enum).

Swift compiler performs exhaustiveness checks for switch statements, which becomes problematic when performed on C enums, where expectations about exhaustiveness are different.

From Swift's point of view, C enums come in two flavors: closed AKA frozen, and open AKA non-frozen. This distinction is aimed at supporting library evolution and ABI stability, while allowing the user to ergonomically work with their code. Swift's solution also supports the unusual C enum coding patterns.

Frozen enums have a fixed set of cases (enumerators in C terms). A library vendor can change (add or remove) cases in a frozen enum, however, it will be both ABI-breaking and source-breaking. In other words, there is a guarantee that the set of enum cases that was seen at compile time exactly matches the set of values that an enum variable can carry at runtime. Swift performs an exhaustiveness check for switch statements on frozen enums: if switch does not handle all enum cases, the user gets a warning. Moreover, the optimizer can make an assumption that a variable that has a frozen enum type will only store values that correspond to enum cases visible at compile time; unused bit patterns can be reused for other purposes.

Non-frozen enums have an extensible set of cases. A library vendor can add cases without breaking ABI or source compatibility. Swift performs a different flavor of exhaustiveness check for switch statements on non-frozen enums: it always requires an `@unknown default` clause, but only produces a warning if the code does not handle all cases available at the compilation time.

```
// C header.

// An open enum: we expect to add more kinds of input devices in future.
enum InputDevice {
  Keyboard,
  Mouse,
  Touchscreen,
} __attribute__((enum_extensibility(open)));

// A closed enum: we think we know enough about the geometry of Earth to
// confidently say that these are all cardinal directions we will ever need.
enum CardinalDirection {
  East,
  West,
  North,
  South,
} __attribute__((enum_extensibility(closed)));
```

```
// C header imported in Swift.

enum InputDevice: CUnsignedInt, Hashable, RawRepresentable {
  init?(rawValue: CUnsignedInt)
  var rawValue: CUnsignedInt { get }
  typealias RawValue = CUnsignedInt
  case Keyboard
  case Mouse
  case Touchscreen
}

@frozen
enum CardinalDirection: CUnsignedInt, Hashable, RawRepresentable {
  init?(rawValue: CUnsignedInt)
  var rawValue: CUnsignedInt { get }
  typealias RawValue = CUnsignedInt
  case East
  case West
  case North
  case South
}
```

C enums that are not marked as open or closed are mapped to structs since Swift 1.0. At that time, we realized that mapping C enums to Swift enums is not correct if the Swift compiler makes Swift-like assumptions about such imported enums. Specifically, Swift compiler assumes that the only allowed bit patterns of an enum value are declared in enum's cases. This assumption is valid for frozen Swift enums (which were the only flavor of enums in Swift 1.0). However, this assumption does not hold for C enums, for which any bit pattern is a valid value; this assumption was creating an undefined behavior hazard. To resolve this issue in Swift 1.0, C enums were imported as Swift structs by default, and their enumerators were exposed as global computed variables. Objective-C enums declared with `NS_ENUM` were assumed to have "enum nature" and were imported as Swift enums.

The concept of open enums was added in Swift 5 (SE-0192 Handling Future Enum Cases), but that proposal did not change the importing strategy of non-annotated C enums, in part because of source compatibility concerns. It might be still possible to change C enums to be imported as open Swift enums, but as the time passes, it will be more difficult to change.

Another feature of C enums is that they expose integer values to the user; furthermore, enum values are implicitly convertible to integers. Swift enums are opaque by default. When imported in Swift, C enums conform to the `RawRepresentable` protocol, allowing the user to explicitly convert between numeric and typed values.

```
// Converting enum values to integers and back.

var south: CardinalDirection = .South
// var southAsInteger: CUnsignedInt = south // error: type mismatch
var southAsInteger: CUnsignedInt = south.rawValue // = 3
var southAsEnum = CardinalDirection(rawValue: 3) // = .South
```

Typedefs

C typedefs are generally mapped to Swift typealiases, except for a few common C coding patterns that are handled in a special way.

```
// C header.

// An ordinary typedef.
typedef int Money;

// A special case pattern that is mapped to a named struct.
typedef struct {
  int x;
  int y;
} Point;
```

```
// C header imported in Swift.

typealias Money = Int

struct Point {
  var x: CInt { get set }
  var y: CInt { get set }
  init()
  init(x: CInt, y: CInt)
}
```

Macros

C macros are generally not imported in Swift. Macros that define constants are imported as readonly variables.

```
// C header.

#define BUFFER_SIZE 4096
#define SERVER_VERSION "3.14"
```

```
// C header imported in Swift.

var BUFFER_SIZE: CInt { get }
var SERVER_VERSION: String { get }
```

See also Apple's documentation about this topic: Using Imported C Macros in Swift.