Accessing Cross Platform Native APIs from Swift

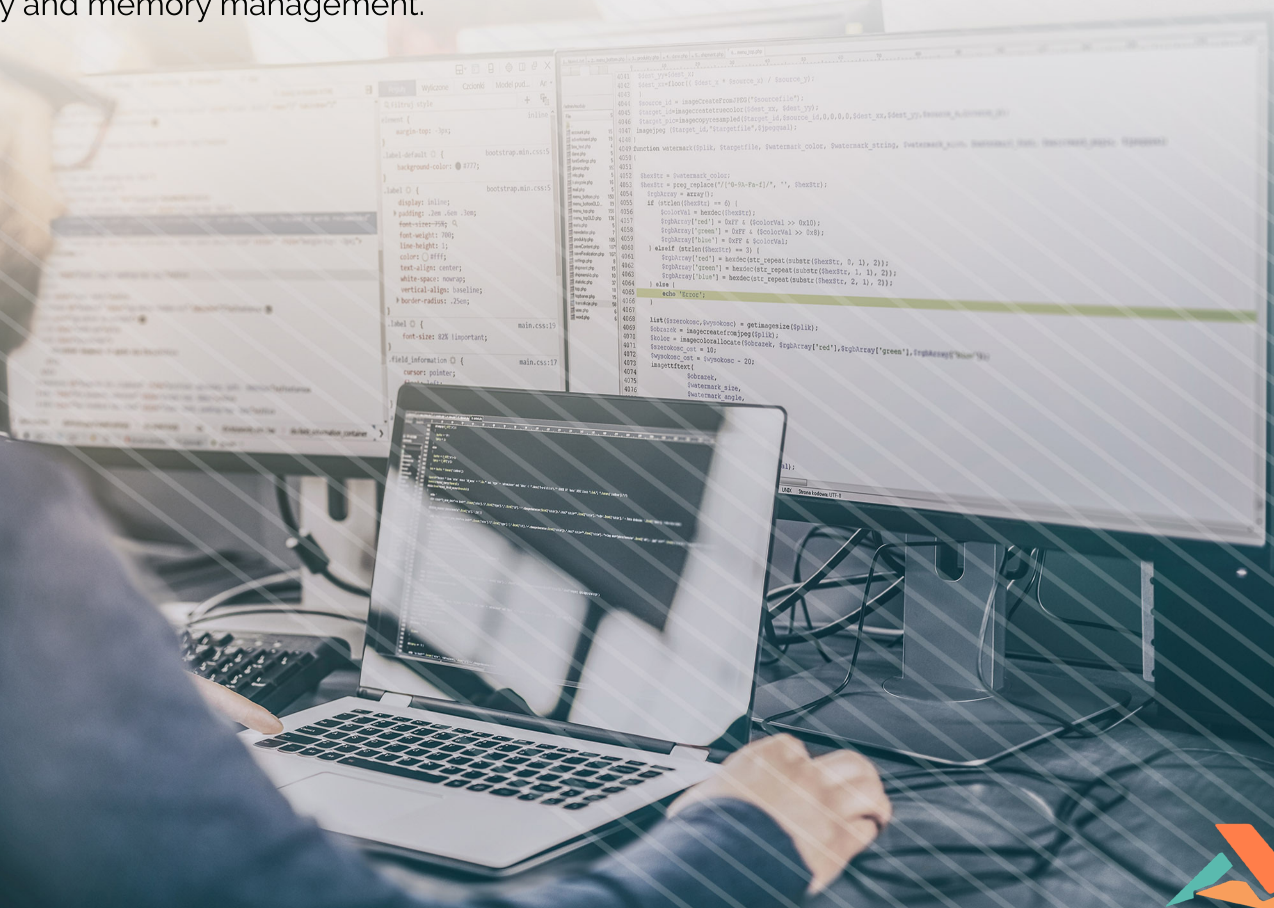
AUTHOR
Daniel Rabiega
Software Engineer II, Barcode Xpress

Apple's Swift programming language has excellent support for working with and consuming native Objective-C APIs. That language, however, has little to no applicability outside of Apple's platforms. You are likely to find that you need to rely on lower level interoperability features in order to interface with cross platform APIs which are most commonly implemented in C or C++.

C is usually the greatest common denominator between whatever language your API is or will be written in and Swift. If you want to consume an existing C API, or are implementing a new one which can be used from C, you will find that the process is considerably simpler than consuming an existing API designed for C++ or some other high level language. Calls to such higher level languages will need to be wrapped in a layer of C using whatever interoperability features are available there.

Even Objective-C APIs can sometimes find this step necessary. Apple's compiler will generally allow you to include C++ features and syntax in Objective-C source code, but it does so by considering such files to be written in an additional language which you might be unaware you are even using: Objective-C++. For most purposes the difference is transparent, but the Swift compiler is not compatible with this superset of Objective-C and you will need to generate header files for your API which do not include anything outside of the base Objective-C specification.

Once you have access to C compatible header files for your API there are two main factors that you will have to contend with to use them from Swift or to create an idiomatic Swift layer: type compatibility and memory management.



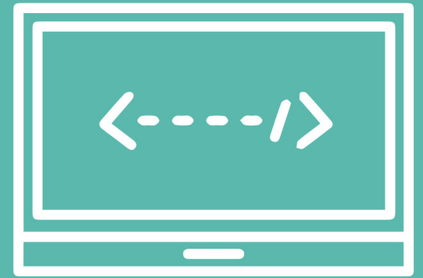
Integral Types

Integers are among the simplest types in any programming language, but there are a dazzling array of integer types. Many of these types have different widths in different platforms and contexts.

If you are using Swift 6 or later, you can utilize a set of C interoperability types which are now included in the Swift standard library. Additional documentation can now be found at that location on some of the other topics covered here. Swift's Int and UInt types will be the same widths as C's long and unsigned long in most cases but you should usually prefer to use the C interoperability types in Swift when available.

An even better option is to not use types whose widths are implementation defined. Swift presents a set of integer types with exact lengths which are detailed below, along with their counterparts from C's stdint.h header.

Swift Type	C / C++ Type (stdint.h)
Int8	Int8_t
Int16	Int16_t
Int32	Int32_t
Int64	Int64_t
UInt8	UInt8_t
UInt16	UInt16_t
UInt32	UInt32_t
UInt64	UInt64_t



Enumerations

Enumerations are ever present in APIs for good reason. Swift code can make use of C enumerations in included headers natively, but both C++ and Swift provide fuller featured enumeration types. These types offer better safety guarantees and a wider variety of backing types. Writing corresponding enum definitions in respective files is relatively straight forward:

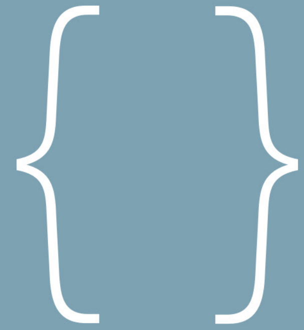
barcode-type.swift

```
public enum SwiftBarcodeType: UInt64 {
  case Barcode1D = 0
  case Barcode2D = 1
  case BarcodePostal = 2
}
```

barcode-type.hpp

```
enum class CppBarcodeType: uint64_t
{
  case Barcode1D = 0
  case Barcode2D = 1
  case BarcodePostal = 2
};
```

Enumerations (cont)



barcode-type.h
typedef CBarcodeType uint64_t;

You can then convert Swift enumeration values back and forth between UInt64's like this:

```
public func getSwiftEnum(val: UInt64)->SwiftBarcodeType {
  return SwiftBarcodeType(rawValue: val)
}
public func getRawValue(val: SwiftBarcodeType)->UInt64 {
  return val.rawValue
}
```

And similarly in C++:

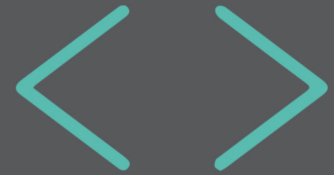
```
CppBarcodeType getCppEnum (uint64_t val) {
  return CppBarcodeType{val};
}
uint64_t getRawValue(CppBarcodeType val){
  Return static_cast<uint64_t>(val);
}
```

Structs and Pointers

Basic structs defined in C header files will be usable as is and can be passed to and from Swift code as values. If you need to pass pointers to and from C APIs, or if your structs contain pointers, Swift will do its best to translate them into one of its equivalent types, of which there are more than a few.

Opaque Pointers

Pointers to types which are declared in your C headers but not defined will be translated by Swift into its OpaquePointer class. Without the definitions present there is not much you can do to them in Swift other than store them and pass them back to the API.



Unsafe Pointers

Pointers to objects which Swift can represent (such as primitives or structs) will be represented in Swift by a set of generics, UnsafePointer<> and UnsafeMutablePointer<>. When pointing to C arrays of these objects, there are buffer versions of each of them: UnsafeBufferPointer<> and UnsafeMutableBufferPointer<>. Finally, all four of these types have an associated "raw" version which is not generic and is equivalent to C's void*s: UnsafeRawPointer, etc.

Swift Pointer Type	Equivalent C Type
UnsafePointer<MyType> UnsafeBufferPointer<MyType>	const MyType*
UnsafeMutablePointer<MyType> UnsafeMutableBufferPointer<MyType>	MyType*
UnsafeRawPointer UnsafeRawBufferPointer	const void*
UnsafeRawMutablePointer UnsafeRawMutableBufferPointer	void*

These types will allow you to work with memory allocated in C or C++ and passed to Swift, but no automatic memory management or garbage collection will be performed on the memory they point to. They are essential for passing data from native code to Swift but you should be careful to make sure that any objects are deallocated after you are done with them.

Special Cases

Swift provides several special cases for passing objects to C APIs that make some common situations considerably easier:

- Swift strings are automatically converted when passed to functions declared as accepting char* arguments.
- Swift arrays can be passed to functions expecting pointer arguments of the equivalent type.
- Swift functions and closures can be passed to C API functions expecting function pointer arguments.

In the cases of arrays and functions you will of course have to ensure that their component types are compatible with what the C function expects.

Passing Memory Managed Objects to C

As mentioned above, there are several ways to pass pointers to Swift objects to C, but this is where some additional issues with memory management come into play. When passing objects which are dynamically allocated, you run the very real risk of passing data into your API which will no longer exist by the time it is used. Thankfully, the Unmanaged class provides some tools for these situations.

Unmanaged provides two static functions for creating instances from Swift objects: Unmanaged.passRetained() and Unmanaged.passUnretained(). The difference between the two is how the memory manager deals with the object your are creating the pointer from. Use passRetained if the C API will need to hold onto the reference for later use, but be warned that you will eventually need to manually release the object.

Once you have created an Unmanaged instance from either of these functions, you can use its toOpaque() method to yield an UnsafeMutableRawPointer which can be passed to a C function.

About Accusoft

Accusoft is a software development company specializing in content processing, conversion, and automation solutions. From out-of-the-box and configurable applications to APIs built for developers, we help organizations solve their most complex content workflow challenges. Our patented solutions enable users to gain insight from content in any format, on any device with greater efficiency, flexibility, and security.