The Peril of the Ampersand

This thread has been locked. Questions are automatically locked after two months of inactivity, or sooner if deemed necessary by a moderator.

151

A recent version of Xcode (1) added new warnings that help detect a ***** gotcha related to the lifetime of unsafe pointers. For example:

- `Initialization of 'UnsafeMutablePointer<timeval>' results in a dangling pointer`
- `Inout expression creates a temporary pointer, but argument 'iov_base' should be a pointer that outlives the call to 'init(iov_base:iov_len:)'`

I've seen a lot of folks confused by these warnings, and by the lifetime of unsafe pointers in general, and this post is my attempt to clarify the topic.

If you have questions about any of this, please put them in a new thread and tag that with *Swift* and *Debugging*.

Finally, I encourage you to watch the following WWDC presentations:

- WWDC 2020 Session 10648 Unsafe Swift
- WWDC 2020 Session 10167 Safely manage pointers in Swift

These cover some of the same ground I've covered here, and a lot of other cool stuff as well.

Share and Enjoy
—
Quinn "The Eskimo!"
Apple Developer Relations, Developer Technical Support, Core OS/Hardware
`let myEmail = "eskimo" + "1" + "@apple.com"`

(1) Swift 5.2.2, as shipped in Xcode 11.4. See the discussion of SR-2790 in Xcode 11.4 Release Notes.

Basics

In Swift, the ampersand (`&`) indicates that a parameter is being passed `inout`. Consider this example:

```
1 func addVarnish(_ product: inout String) {
2     product += " varnish"
3 }
4
5 var waffle = "waffle"
6 addVarnish(&waffle)
7 print(waffle)
8 // printed: waffle varnish
```

On line 6, the ampersand tells you that `waffle` could be modified by `addVarnish(_:)`.

However, there is another use of ampersand that was designed to help with C interoperability. Consider this code:

```
var tv = timeval()
gettimeofday(&tv, nil)
print(tv)
// printed: timeval(tv_sec: 1590743104, tv_usec: 77027)
```

The first parameter to `gettimeofday` is an `UnsafeMutablePointer<timeval>`. Here the ampersand denotes a conversion from a `timeval` to an `UnsafeMutablePointer<timeval>`. This conversion makes it much easier to call common C APIs from Swift.

This also works for array values. For example:

```
var hostName = [CChar](repeating: 0, count: 256)
gethostname(&hostName, hostName.count)
print(String(cString: hostName))
// printed: slimey.local.
```

In this code the ampersand denotes a conversion from `[CChar]` to an `UnsafeMutablePointer<CChar>` that points to the base of the array.

While this is convenient, it's potentially misleading, especially if you come from a C background. In C-based languages, using ampersand in this way yields a pointer to the value that's valid until the value gets deallocated. That's not the case in Swift. Rather, **the pointer generated by the ampersand syntax is only valid for the duration of that function call**.

To understand why that's the case, consider this code:

```
 1 struct TimeInTwoParts {
 2     var sec: time_t = 0
 3     var usec: Int32 = 0
 4     var combined: timeval {
 5         get { timeval(tv_sec: sec, tv_usec: usec) }
 6         set {
 7             sec = newValue.tv_sec
 8             usec = newValue.tv_usec
 9         }
10     }
11 }
12 var time = TimeInTwoParts()
13 gettimeofday(&time.combined, nil)
14 print(time.combined)
15 // printed: timeval(tv_sec: 1590743484, tv_usec: 89118)
16 print(time.sec)
17 // printed: 1590743484
18 print(time.usec)
19 // printed: 89118
```

Here `combined` is a computed property that has no independent existence in memory. Thus, it simply makes no sense to take the address of it.

So, how does ampersand deal with this? Under the covers the Swift compiler expands line 13 to something like this:

```
var tmp = time.combined
gettimeofday(&tmp, nil)
time.combined = tmp
```

Once you understand this it's clear why the resulting pointer is only valid for the duration of the call: As soon as Swift cleans up `tmp`, the pointer becomes invalid.

A Gotcha

This automatic conversion can be a ***** gotcha. Consider this code:

```
1 var tv = timeval()
2 let tvPtr = UnsafeMutablePointer(&tv)
3 //          ^~~~~~~~~~~~~~~~~~~~~~~~~
4 // Initialization of 'UnsafeMutablePointer<timeval>' results in a dangling pointer
5 gettimeofday(tvPtr, nil)
```

This results in undefined behaviour because the pointer generated by the ampersand on line 2 is no longer valid when it's used on line 5. In some cases, like this one, the later Swift compiler is able to detect this problem and warn you about it. In other cases you're not so lucky. Consider this code:

```
1 guard let f = fopen("tmp.txt", "w") else { … }
2 var buf = [CChar](repeating: 0, count: 1024)
3 setvbuf(f, &buf, _IOFBF, buf.count)
4 let message = [UInt8]("Hello Crueld World!".utf8)
5 fwrite(message, message.count, 1, f)
6 fclose(f)
```

This uses `setvbuf` to apply a custom buffer to the file handle. The file handle uses this buffer until after the close on line 6. However, the pointer created by the ampersand on line 3 only exists for the duration of the `setvbuf` call. When the code calls `fwrite` on line 5 the buffer pointer is no valid and things end badly.

Unfortunately the compiler isn't able to detect this problem. Worse yet, the code might actually work initially, and then stop working as you change optimisation settings, update the compiler, change unrelated code, and so on.

Another Gotcha

There is another gotcha associated with the ampersand syntax. Consider this code:

```
class AtomicCounter {
    var count: Int32 = 0
    func increment() {
        OSAtomicAdd32(1, &count)
    }
}
```

This looks like it'll implement an atomic counter but there's no guarantee that the counter will be atomic. To understand why, apply the `tmp` transform from earlier:

```
class AtomicCounter {
    var count: Int32 = 0
    func increment() {
        var tmp = count
        OSAtomicAdd32(1, &tmp)
        count = tmp
    }
}
```

So each call to `OSAtomicAdd32` could potentially be operating on a separate copy of the counter that's then assign back to `count`. This undermines the whole notion of atomicity.

Again, this might work in some builds of your product and then fail in other builds.

Summary

So, to summarise:

- Swift's ampersand syntax has very different semantics from the equivalent syntax in C.
- When you use an ampersand to convert from a value to a pointer as part of a function call, make sure that the called function doesn't use the pointer after it's returned.
- It is not safe to use the ampersand syntax for functions where the exact pointer matters.

It's Not Just Ampersands

There's one further gotcha related to arrays. The `gethostname` example above shows that you can use an ampersand to pass the base address of an array to a function that takes a mutable pointer. Swift supports two other implicit conversions like this:

- From `String` to `UnsafePointer<CChar>` — This allows you to pass a Swift string to an API that takes a C string. For example:

```
    let greeting = "Hello Cruel World!"
    let greetingLength = strlen(greeting)
    print(greetingLength)
    // printed: 18

```

- From `Array<Element>` to `UnsafePointer<Element>` — This allows you to pass a Swift array to a C API that takes an array (in C, arrays are typically represented as a base pointer and a length). For example:

```
    let charsUTF16: [UniChar] = [72, 101, 108, 108, 111, 32, 67, 114, 117, 101, 108, 32, 87, 111, 114, 
108, 100, 33]
    print(charsUTF16)
    let str = CFStringCreateWithCharacters(nil, charsUTF16, charsUTF16.count)!
    print(str)
    // prints: Hello Cruel World!

```

Note that there's no ampersand in either of these examples. This technique only works for `UnsafePointer` parameters (as opposed to `UnsafeMutablePointer` parameters), so the called function can't modify its buffer. As the ampersand is there to indicate that the value might be modified, it's not used in this immutable case.

However, the same pointer lifetime restriction applies: The pointer passed to the function is only valid for the duration of that function call. If the function keeps a copy of that pointer and then uses it later on, Bad Things™ will happen.

Consider this code:

```
func printAfterDelay(_ str: UnsafePointer<CChar>) {
    print(strlen(str))
    // printed: 18
    DispatchQueue.main.asyncAfter(deadline: .now() + 1.0) {
        print(strlen(str))
        // printed: 0
    }
}
let greeting = ["Hello", "Cruel", "World!"].joined(separator: " ")
printAfterDelay(greeting)
dispatchMain()
```

The second call to `strlen` yields undefined behaviour because the pointer passed to `printAfterDelay(_:)` becomes invalid once `printAfterDelay(_:)` returns. In this specific example the memory pointed to by `str` happened to contain a zero, and hence `strlen` returned 0 but that's not guaranteed. The `str` pointer is dangling, so you might get any result from `strlen`, including a crash.

Advice

So, what can you do about this? There's two basic strategies here:

- Extend the lifetime of the pointer
- Manual memory management

Extending the Pointer's Lifetime

The first strategy makes sense when you have a limited number of pointers and their lifespan is limited. For example, you can fix the `setvbuf` code from above by changing it to:

```
let message = [UInt8]("Hello Crueld World!".utf8)
guard let f = fopen("tmp.txt", "w") else { … }
var buf = [CChar](repeating: 0, count: 1024)
buf.withUnsafeMutableBufferPointer { buf in
    setvbuf(f, buf.baseAddress!, _IOFBF, buf.count)
    fwrite(message, message.count, 1, f)
    fclose(f)
}
```

This version of the code uses `withUnsafeMutableBufferPointer(_:)`. That calls the supplied closure and passes it a pointer (actually an `UnsafeMutableBufferPointer`) that's valid for the duration of that closure. As long as you only use that pointer inside the closure, you're safe!

There are a variety of other routines like `withUnsafeMutableBufferPointer(_:)`, including:

- The withUnsafeMutablePointer(to:_:) function
- The withUnsafeBufferPointer(_:), withUnsafeMutableBufferPointer(_:), withUnsafeBytes(_:), and withUnsafeMutableBytes(_:) methods on `Array`
- The withUnsafeBytes(_:) and withUnsafeMutableBytes(_:) methods on `Data`
- The withCString(_:) and withUTF8(_:) methods on `String`.

Manual Memory Management

If you have to wrangle an unbounded number of pointers — or the lifetime of your pointer isn't simple, for example when calling an asynchronous call — you must revert to manual memory management. Consider the following code, which is a Swift-friendly wrapper around `posix_spawn`:

```
func spawn(arguments: [String]) throws -> pid_t {
    var argv = arguments.map { arg -> UnsafeMutablePointer<CChar>? in
        strdup(arg)
    }
    argv.append(nil)
    defer {
        argv.forEach { free($0) }
    }
    var pid: pid_t = 0
    let success = posix_spawn(&pid, argv[0], nil, nil, argv, environ) == 0
    guard success else { throw NSError(domain: NSPOSIXErrorDomain, code: Int(errno), userInfo: nil) }
    return pid
}
```

This code can't use the `withCString(_:)` method on `String` because it has to deal with an arbitrary number of strings. Instead, it uses `strdup` to copy each string to its own manually managed buffer. And, as these buffers are manually managed, is has to remember to free them.

Change History

- 1 Jun 2020 — Initial version.
- 24 Feb 2021 — Fixed the formatting. Added links to the WWDC 2021 sessions. Fixed the feedback advice. Minor editorial changes.

Swift Debugging

Asked 1 week ago by eskimo

Reply to this question

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

Copyright © 2021 Apple Inc. All rights reserved.