Objective-C Memory Management for Swift Programmers

This thread has been locked by a moderator.

If you're new to Apple platforms (welcome!) then you may encounter memory management terms that don't fit into the Swift world view of strong and weak references. This post is my attempt to explain that terminology from a historical perspective.

If you have a question about this stuff, please start a new thread with the *Objective-C Runtime* tag and I'll try to respond there.

ps This is written from an Apple perspective. Objective-C had a long history prior to Apple's acquisition of NeXT. I wasn't part of that story, so I'm not going to attempt to summarise it here.

Share and Enjoy

—

Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Objective-C Memory Management for Swift Programmers

In the beginning there was manual retain/release (MRR). Here's a snippet:

```
1 - (void)printWaffleWithVarnish:(NSString *)varnish {
2     NSMutableString * waffle = [[NSMutableString alloc] init];
3     [waffle appendString:@"waffle "];
4     [waffle appendString:varnish];
5     … print waffle …
6     [waffle release];
7 }
```

Yes, this is weird Objective-C syntax but lemme walk you through it:

- Line 1 defines a method called `-printWaffleWithVarnish:` that takes a single string parameter, `varnish`.
- Line 2 allocates a `waffle` mutable string. This alloc/init sequence returns an object with a +1 reference count.
- Lines 3 through 5 work with that.
- Line 6 releases the reference to it. This balances the +1 reference acquired in line 2.

This model has one *really* nice property, namely that all the reasoning is local:

- The person implementing `-printWaffleWithVarnish:` doesn't have to worry about the memory management for `varnish`. It's the caller's responsibility to make sure that it's allocated before calling the method and deallocated after.
- The person calling `-printWaffleWithVarnish:` doesn't have to worry about the memory management within that method.

So, you can look at a method in isolation and know whether the memory management is correct.

Enter Autorelease

The above is fine but what do you do if you want a method that returns an object? You could always require methods to return objects with a +1 reference count, but that's a hassle and, worse yet, it undermines the local reasoning property. To know whether the memory management is correct you have to look at both the method and its callers.

The solution here is autorelease. Consider this method:

```
1 - (NSString *)waffleWithVarnish:(NSString *)varnish {
2     NSMutableString * waffle = [[NSMutableString alloc] init];
3     [waffle appendString:@"waffle "];
4     [waffle appendString:varnish];
5     return [waffle autorelease];
6 }
```

The method definition on line 1 indicates that this returns a string with a +0 reference count. But… but… but… how is *that* possible?

Well, the `-autorelease` on line 5 is the secret. It tells the runtime to add the object to the **autorelease pool**. At some point in the future the runtime will drain the pool. In the meantime, however, the person who called `-waffleWithVarnish:` is free to use the object. And if they want to hold on to the object for a long period of time, they can retain it themselves.

This design has a number of nice features: It maintains the local reasoning property, and it's easy for both the method's implementer and the method's caller.

Note You can think of the autorelease pool as a giant array. When you call `-autorelease`, it adds the object to the array (without incrementing the retain count). When you drain the pool, it empties the array, releasing all the references as it goes. Back in the day the autorelease pool implementation was actually that simple. On modern systems it's much more complex, with the benefit that it's also *much* faster.

So, when does this pool drain? Well, Objective-C's bread and butter is GUI frameworks, and in a GUI framework there's an obvious place to drain the pool, namely the event loop. Every time your app returns back to the event loop, the runtime drains the pool.

Manually Draining the Pool

This memory management model is good, but it's not perfect. This code illustrates a common gotcha:

```
NSArray * varnishes = … a very large array of varnishes …
for (NSString * varnish in varnishes) {
    NSString * waffle = [waffleFactory waffleWithVarnish:varnish];
    … work with the waffle …
}
```

Every time it calls `-waffleWithVarnish:`, the returned object ends up in the autorelease pool. If there are a lot of varnishes to apply, there can be a lot of objects in the pool. Those objects aren't leaked — they'll eventually be released when the pool drains — but they do increase the peak memory consumption of your code.

The solution is to add a local autorelease pool:

```
NSArray * varnishes = … a very large array of varnishes …
for (NSString * varnish in varnishes) {
    @autoreleasepool {
        NSString * waffle = [waffleFactory waffleWithVarnish:varnish];
        … work with the waffle …
    }
}
```

This drains the pool each time around the loop, avoiding any build up.

These autorelease pools form a stack. The `@autoreleasepool { }` construct pushes a pool on to the stack, runs the code inside the curly brackets, then pops the pool off the top of the stack and drains it. When you autorelease an object, it's always added to the pool on top of the stack.

The `@autoreleasepool { }` construct arrived relatively late in the Objective-C story. Before that folks managed the autorelease pool stack with the `NSAutoreleasePool` type.

This use of autorelease pools inside tight loops is one reason why it's really important that the implementation be fast. Historically that wasn't the case, and so you might see code that attempts to amortise the cost of pushing and popping the pool by unrolling the loop. Thank goodness we don't have to do that any more (-:

Modern Objective-C

The local reasoning property of MRR makes it feasible to create a tool to check that the code is managing memory correctly. In Xcode 3.2 Apple added the static analyser which, amongst other things, can detect the most common bugs in MRR code.

The success of the static analyser prompted Apple to introduce automatic reference counting (ARC) in Xcode 4.2. In this model the compiler adds the necessary memory management calls for you.

Enter Swift, Stage Left

So far, so Objective-C. It seems like none of the above is relevant to Swift programmers but it turns out that's not the case. Imagine a Swift program like this:

```
let varnishes: [String] = … a very large array of varnishes …
for varnish in varnishes {
    let waffle = waffleFactory.waffle(with: varnish)
    … work with the waffle …
}
```

If the waffle factory is implemented in Objective-C, this can result in objects building up inside the autorelease pool. You solve this problem in Swift the same way you do in Objective-C, by adding an autorelease pool within the loop:

```
let varnishes: [String] = … a very large array of varnishes …
for varnish in varnishes {
    autoreleasepool {
        let waffle = waffleFactory.waffle(with: varnish)
        … work with the waffle …
    }
}
```

Swift itself does not use the autorelease pool. This only happens when you interact with Objective-C. Occasionally I see this bite folks bringing code from other platforms to an Apple platform. For example, if you have code that calls `FileHandle` in a tight loop, it won't need an `autoreleasepool(_:)` call on Linux but it will on macOS.

Threads

Every thread has its own stack of autorelease pools. This makes sense when you think about it. If you're running the above code on a secondary thread you don't want your objects being released when the main thread returns to its event loop.

Back in the day you had to set this up manually for secondary threads (the GUI framework would set it up for the main thread). If you didn't, your thread would run with an empty autorelease pool stack and any objects you autoreleased ended up being leaked!

At some point (macOS 10.7?) the system started creating an autorelease pool stack for every new thread. However, there's still a gotcha here. Consider this code:

```
func nextVarnish() -> String {
    … read varnish from the network …
}

Thread.detachNewThread {
    let waffleFactory = WaffleFactory()
    while true {
        let varnish = nextVarnish()
        let waffle = waffleFactory.waffle(with: varnish)
        … work with the waffle …
    }
}
```

The thread never terminates and so the autorelease pool never drains. This is what we call **abandoned memory**. It's not a leak, because the process still has a reference to the objects in the pool and could theoretically release them, but that never happens in practice.

The fix is to add an autorelease pool:

```
Thread.detachNewThread {
    let waffleFactory = WaffleFactory()
    while true {
        autoreleasepool {
            let varnish = nextVarnish()
            let waffle = waffleFactory.waffle(with: varnish)
            … work with the waffle …
        }
    }
}
```

Dispatch

Originally Dispatch knew nothing about autorelease pools. This meant that any work item you ran on a queue, other than the main queue, needed to set up and drain an autorelease pool. Without this it would leak. Yikes!

This got worse when we added a default autorelease pool to every thread. This included the worker threads used by Dispatch. Now autoreleased objects wouldn't leak, but rather build up in the worker thread's autorelease pool. These would only be released when Dispatch terminated the worker thread, which in many cases never happened. Double yikes!

This was fixed in macOS 10.12, where Dispatch finally got some autorelease pool smarts. There is now a pool associated with each queue and you can customise the frequency that it drains use an `AutoreleaseFrequency` value.

Crashing Out of the Pool

As a Swift programmer, your first encounter with the autorelease pool might be a crash. Consider this Swift function:

```
1 func simulateBadMRR() {
2     let b = (0...1023).map { _ in UInt8.random(in: 0...255) }
3     let d = NSData(bytes: b, length: b.count)
4     let du = Unmanaged.passUnretained(d)
5     print(du.takeUnretainedValue().count)
6     _ = du.autorelease()
7 }
```

This simulates someone writing some bad MRR code. Line 4 creates a +0 reference to the `NSData` object and line 6 erroneously autoreleases that reference.

If you call this function from Swift then, unless you're very lucky, you won't crash in the function itself. Remember that the function added its `NSData` reference to the autorelease pool, so the erroneous extra release doesn't happen here. Rather, the function returns, and then your code returns, and eventually the thread gets back to a point where the system drains the autorelease pool. This finally performs the erroneous release and your program crashes.

In the crash report you'll see something like this:

```
0 libobjc.A.dylib        … objc_release + 42
1 libobjc.A.dylib        … AutoreleasePoolPage::releaseUntil(objc_object**) + 168
2 libobjc.A.dylib        … objc_autoreleasePoolPop + 227
3 CoreFoundation         … _CFAutoreleasePoolPop + 22
4 Foundation             … -[NSAutoreleasePool drain] + 133
5 AppKit                 … -[NSApplication run] + 636
6 AppKit                 … NSApplicationMain + 817
7 CrashingOutOfThePool   … main + 9 (AppDelegate.swift:4)
8 dyld                   … start + 2432
```

This example is from an AppKit app, and so frame 5 is the `NSApplication.run()` method draining the autorelease pool. Frames 4 through 1 are autorelease pool gubbins, where frame 1 actually performs the erroneous release which triggers the crash in frame 0.

Debugging *over release* bugs like this is hard because the source of the error, the extraneous autorelease in `simulateBadMRR()`, is far removed from the crash. Fortunately, Apple has your back here, in the form of the zombies feature. This does two things:

- It replaces deallocated objects with a zombie. This traps if anyone calls it, which makes crashes like this easy to reproduce. For example, in my test app I had a button wired up to the `simulateBadMRR()` code and the first time I clicked that button it didn't crash. I had to click it twice to trigger the crash. With zombies enabled, it traps on the first click.
- In Instruments the Zombies template enables reference count tracking, so you get a backtrace of where the object was allocated and every time its reference count was modified.

For more information about the zombies feature, see my Standard Memory Debugging Tools post.

Revision History

- **2023-03-23** Added the *Crashing Out of the Pool* section.
- **2022-09-28** First posted.

Swift Objective-C Runtime

Reply

Posted 7 months ago by eskimo

Add a Comment