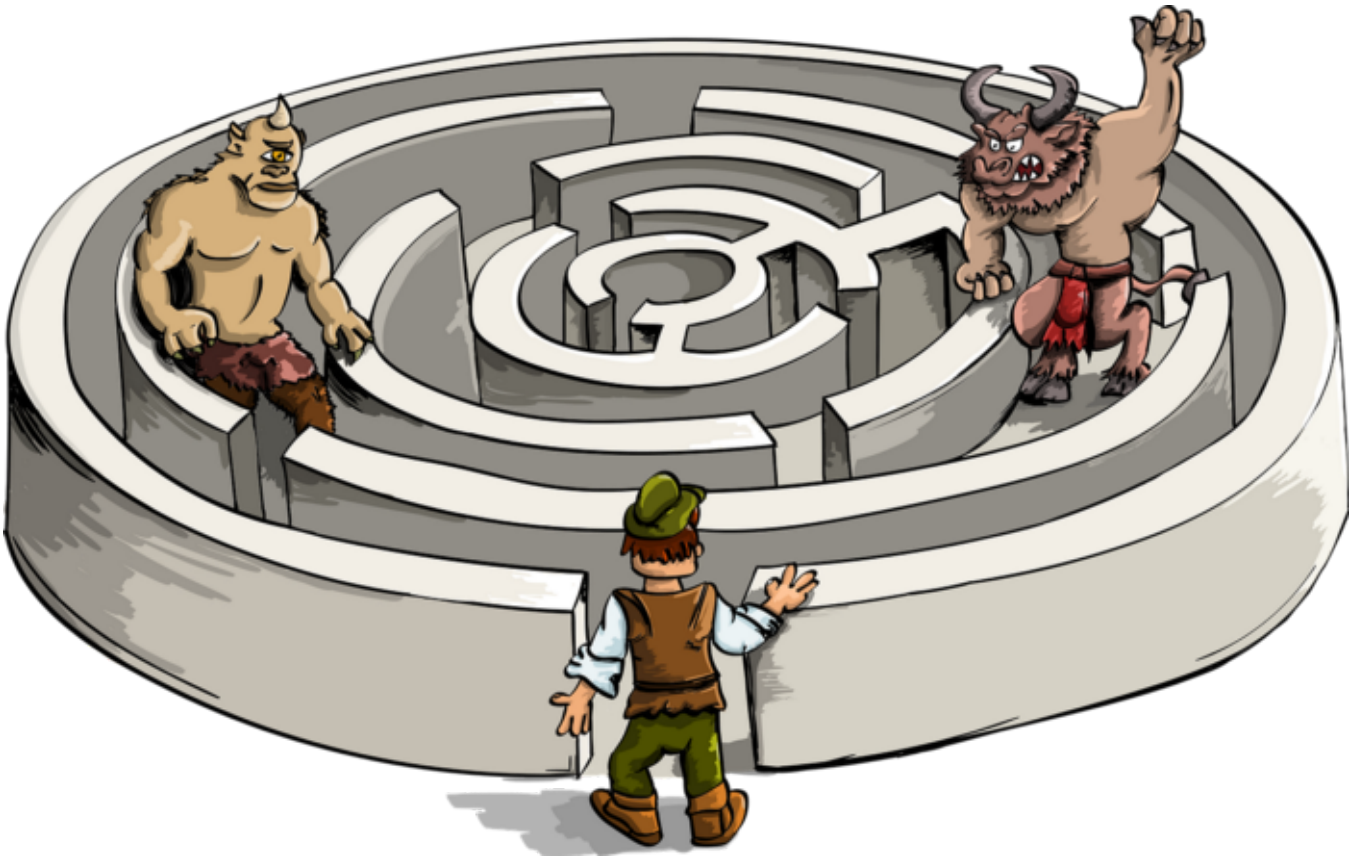
When self should be weakify?

How to diagnose retain cycle and when to use [weak self] to avoid retain cycle.

Aaina jain
Jan 13 · 4 min read ★



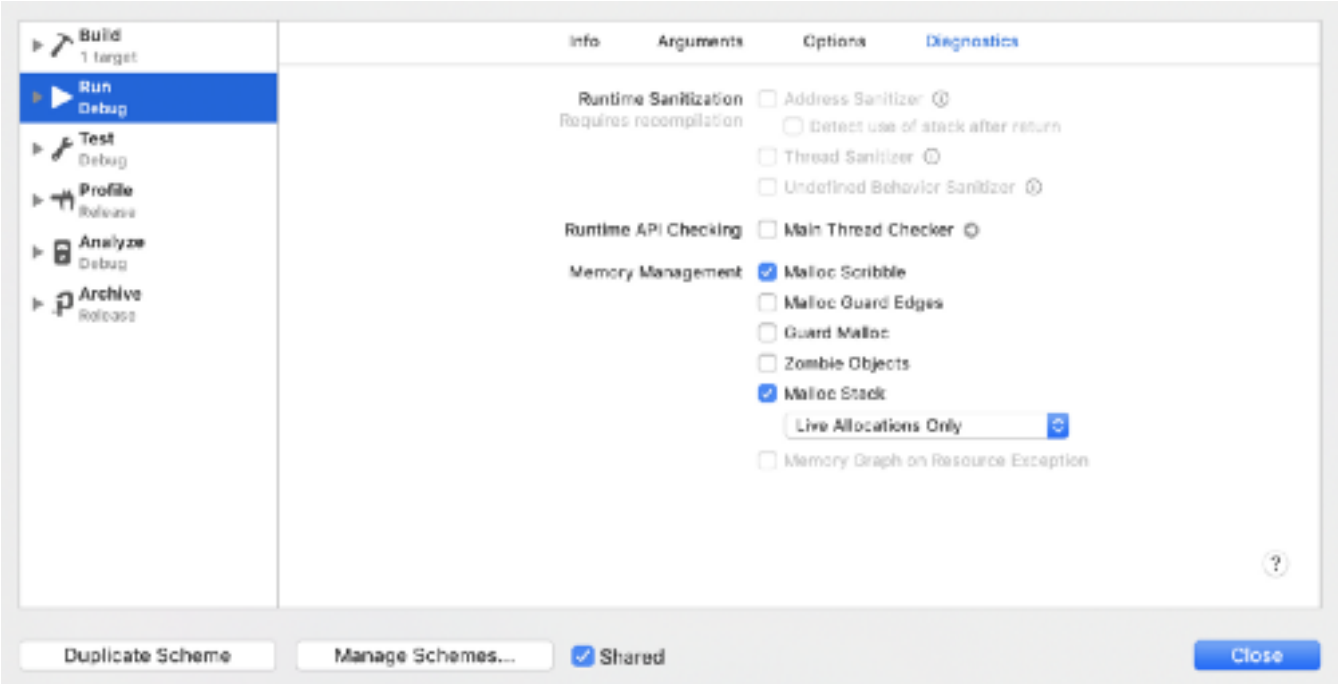
Credit: https://pixabay.com/

Introduction

Often we end up having a memory leak in the project. We know when our closure can increase retain count and when not. When I was executing function [Using self inside] in Notification Center Observer, I had an intuition that it won't cause a retain cycle. As I was not calling `NotificationCenter` on self so I was pretty sure that it won't cause a retain cycle but it did 😰.

How to diagnose retain cycle:

- *Debug Memory Graph:* To use debug memory graph, first you need to edit scheme.



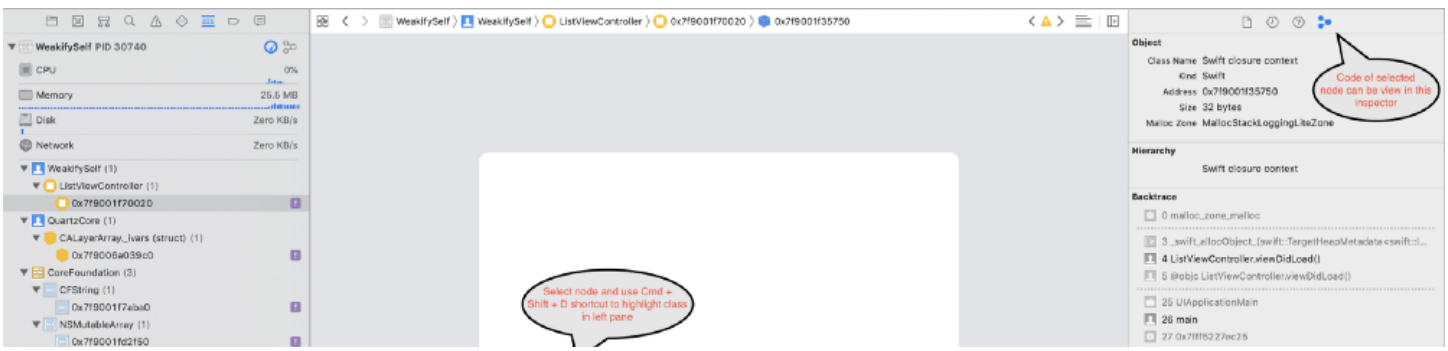
Malloc scribble will show you code in Show Memory Inspector when `Debug Memory Graph` is launched.

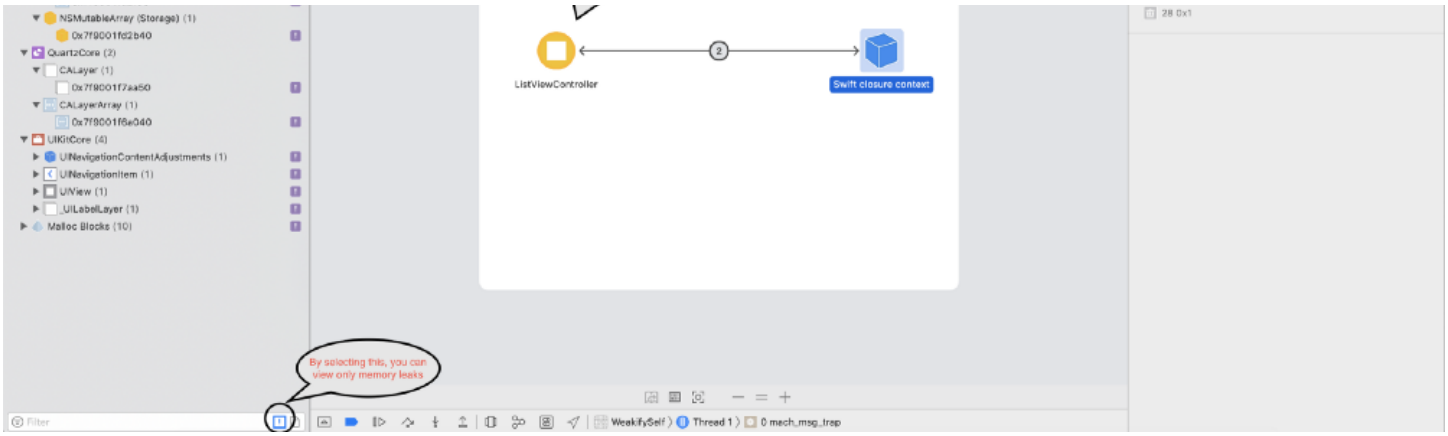
You can tap on show only leaked blocks to see only leaked classes.

- Perform whatever app actions you want to analyse (the actions which can cause retain cycles).

Open memory graph debugging mode by selecting this button.

- The memory graph debugger pauses app execution and shows the following:





- *Or you can put breakpoint in* `deinit` *to check class has been deinitialized or not.*

Let's take some use cases which can cause retain cycle:

Use Cases:

1. Assigning closure to closure:

While using any instance property or function inside closure, compile gives us the warning to use `self`.

```
listTitle = {
    print(view.debugDescription)
    print("closure executed
    return "Fiction Books"
}
```

Implicit use of 'self' in closure; use 'self.' to make capture semantics explicit

Now, while using `self` we need to make sure that it should be weak else it will cause a retain cycle.

Note: `lazy` closure can also cause a retain cycle.

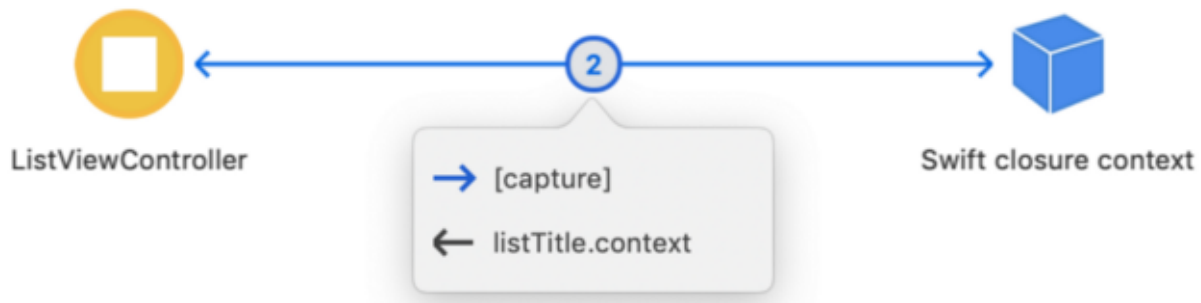
Why will it cause retain cycle: Because closure captures variables which will end up in increasing retain count. If we don't use `self` ` inside closure then there is no retain cycle.

```
import Foundation
import UIKit

class ListViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
```

```
        super.viewDidLoad()
        listTitle = {
            print(self.view.debugDescription)
            print("closure executed")
            return "Fiction Books"
        }
    }
}

class DetailViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
        super.viewDidLoad()
        guard let title = listTitle?() else { return }
        self.title = title
    }
}
```

ClosureWithSelf.swift hosted with ❤ by GitHub view raw

https://gist.github.com/aainaj/0acdb879506893991954b68478cec024



Picture from Memory graph shows retain cycle due to closure holding strong self

Link for closure without self:
https://gist.github.com/aainaj/1544b9047a94b0e0015803808d58f3fc

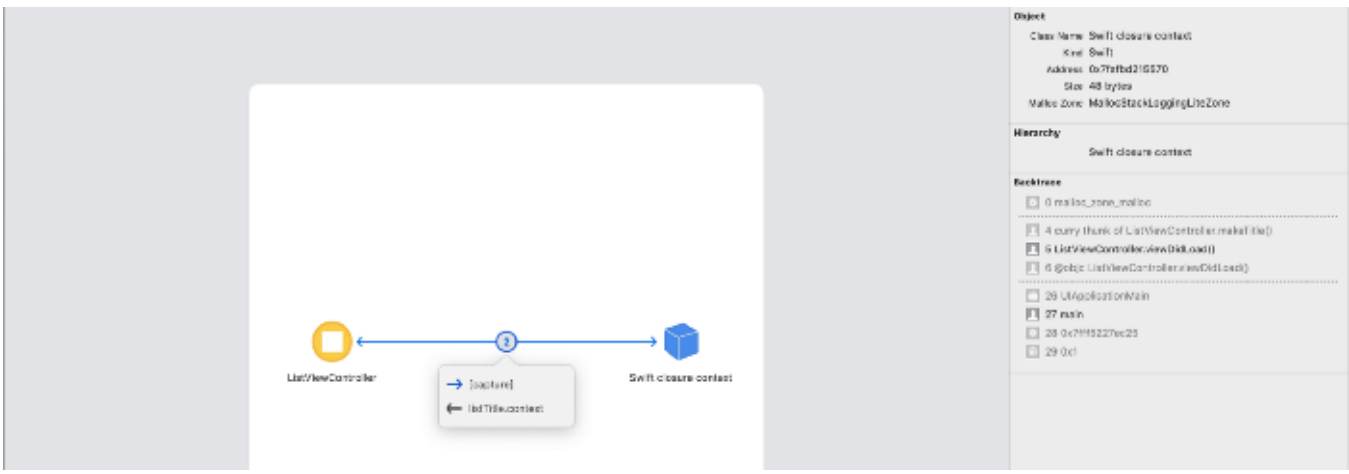
2. Assigning function to closure:

On assigning function to closure, we assign function pointer so it actually. Function pointer is defined as: `ListViewController.makeList(self)` . It results in a retain cycle always. So it's good to always assign `[weak self]` to closure.

```swift
import Foundation
import UIKit

class ListViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
        super.viewDidLoad()
        listTitle = makeTitle
    }

    func makeTitle() -> String {
        print("function executed")
        return "Fiction Books"
    }
}


class DetailViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
        super.viewDidLoad()
        guard let title = listTitle?() else { return }
        self.title = title
    }
}
```

FunctionAssignmentToClosure.swift hosted with ❤ by GitHub view raw

https://gist.github.com/aainaj/baf43fe7afb4f015561dd2bb20f0ecfd



Retain cycle can be avoided by using `[weak self]`

```
listTitle = { [weak self] in
    self?.makeTitle()
}
```

3. DispatchQueue:

Invoking closure in `DispatchQueue` doesn't create a retain cycle as we are not calling this function on `self` `. It's safe to use below code:

```
DispatchQueue.main.asyncAfter(deadline: .now() + 5, execute: {
    self.label.textColor = .green
})
```

4. Async DispatchQueue:

In the previous example, We just changed the text color of a label so we may think that it won't cause a retain cycle. But what if we are doing async API calls in dispatch queue. In this case, it doesn't retain self but finishes all tasks submitted to the queue and then deallocate class. When a view is going off-screen, we can cancel tasks so that they don't get executed when class is not in the hierarchy.

```
import UIKit

class ListViewController: UIViewController {
    let label = UILabel(frame: CGRect(x: 100, y: 200, width: 300, height: 50))

    override func viewDidLoad() {
        super.viewDidLoad()

        view.backgroundColor = .white

        label.text = "Animating"
        view.addSubview(label)
        let controller = DetailViewController()
        controller.definesPresentationContext = true

        let navControll = UINavigationController(rootViewController: controller)
        self.navigationController?.present(navControll, animated: true, completion: nil
    }

    deinit {
        print("List controller Deinit")
    }
}

class DetailViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
        super.viewDidLoad()

        performAsyncTaskIntoConcurrentQueue(with: {
            print("\n############")
            print("###### All images are downloaded")
        })
    }

```

```swift
    func performAsyncTaskIntoConcurrentQueue(with completion: @escaping () -> ()) {
        let queue = DispatchQueue(label: "com.queue.Concurrent", attributes: .concurrer
        let group = DispatchGroup()
        for i in 1...5 {
            group.enter()
            queue.async {
                let imageURL = URL(string: "https://upload.wikimedia.org/wikipedia/comm
                let _ = try! Data(contentsOf: imageURL)
                print(self.view.description)
                print("###### Image \(i) Downloaded ######")
                group.leave()
            }
        }
        group.notify(queue: DispatchQueue.main) {
            completion()
        }
    }

    deinit {
        print("Detail controller got deallocated")
    }
}
```

DispatchQueueAsyncTaskRetainCycle.swift hosted with ❤ by GitHub view raw

https://gist.github.com/aainaj/3a00c4e2959c5ce678d9582004965110

```
<UIView: 0x7fbee005eac0; frame = (0 0; 414 842); autoresize = W+H; layer =
    <CALayer: 0x7fbee005e7c0>>
###### Image 5 Downloaded ######
<UIView: 0x7fbee005eac0; frame = (0 0; 414 842); autoresize = W+H; layer =
    <CALayer: 0x7fbee005e7c0>>
###### Image 2 Downloaded ######
<UIView: 0x7fbee005eac0; frame = (0 0; 414 842); autoresize = W+H; layer =
    <CALayer: 0x7fbee005e7c0>>
###### Image 3 Downloaded ######
List controller Deinit
<UIView: 0x7fbee005eac0; frame = (0 0; 414 842); autoresize = W+H; layer =
    <CALayer: 0x7fbee005e7c0>>
###### Image 1 Downloaded ######
<UIView: 0x7fbee005eac0; frame = (0 0; 414 842); autoresize = W+H; layer =
    <CALayer: 0x7fbee005e7c0>>
###### Image 4 Downloaded ######

#############
###### All images are downloaded
Detail controller got deallocated
```

5. NotificationCenter:

As we don't call notification center on `self` so it shouldn't cause a retain cycle. But `NotificationCenter.addObserver` document says it copies our block so `[weak self]` should be used.

Adds an entry to the notification center's dispatch table that includes a notification queue and a block to add to the queue, and an optional notification name and sender.

Declaration

```
func addObserver(forName name: NSNotification.Name?,
          object obj: Any?,
           queue: OperationQueue?,
           using block: @escaping (Notification) -> Void) -> NSObjectProtocol
```

block

The block to be executed when the notification is received.

The block is copied by the notification center and (the copy) held until the observer registration is removed.

The block takes one argument:

```
1  import UIKit
2
3  class ListViewController: UIViewController {
4     let label = UILabel(frame: CGRect(x: 100, y: 200, width: 300, height: 50))
5
6      override func viewDidLoad() {
7          super.viewDidLoad()
8          view.backgroundColor = .white
9
10         label.text = "Animating"
11         view.addSubview(label)
12         let controller = DetailViewController()
13         controller.definesPresentationContext = true
14         let navControll = UINavigationController(rootViewController: controller)
15         self.navigationController?.present(navControll, animated: true, completion: nil
16
```

```swift
        NotificationCenter.default.addObserver(forName: .titleUpdated, object: nil, que
            UIView.animate(withDuration: 0.3, animations: {
                self.label.textColor = .green
            })
        }
    }

    deinit {
        print("Deinit")
    }
}

class DetailViewController: UIViewController {
    var listTitle: (() -> String)?

    override func viewDidLoad() {
        super.viewDidLoad()
        NotificationCenter.default.post(name: .titleUpdated, object: nil)
    }
}

extension Notification.Name {
    static let titleUpdated = Notification.Name("titleUpdated")
}
```

NotificationCenterRetainCycle.swift hosted with ❤ by GitHub view raw

https://gist.github.com/aainaj/a85baa708374ae47dfbf75a0464e0de0

Retain cycle caused by notification center

6. UIView.animate:

It's not called on `self` so using `self` inside `animations` block doesn't cause a retain cycle.

```
UIView.animate(withDuration: 0.3, animations: {
    self.label.center = CGPoint(x: 200, y: 300)
})
```

Reward:

Credit: pexels.com

Swift Memory Management Retain Cycle iOS