iOS handles memory management with ARC

Automatic Reference Counting (ARC) for memory management. Every time a new object of a class is created, ARC allocates some amount of memory to store data associated with that object. Each object will have a reference count property that keeps track of everything that has a strong reference to it. Every time a strong reference is created, the reference count of that object gets incremented by one. And whenever a reference goes out of scope, the reference count gets decremented by one.

If two objects only refer to each other and have no other references. Since the reference count for both of them cannot be zero, they cannot be deallocated and will continue to occupy memory. This is known as a strong reference cycle or a retain cycle.

For example if I have two classes one called country and another one called city and I create an object of the country class into my city class and I use the class property called cityName that it has a reference to city class in my city function and then run my app this will create a retain cycle and this will lead to a memory leak. To break this cycle, we use weak and unowned references.

To break the retain cycles we make use of weak references. By default, all references in swift are strong and increment the reference count by 1 upon creation. But weak references don’t affect the reference count. Also, weak references are always optional variables and when the reference count becomes zero, weak reference gets set to nil.

For example if my class add weak var cityName: City and when my function is called in the runApp, reference count to City does not increment. So when all other references to city are removed, it gets deallocated and reference count for country becomes zero.

Similar to weak references, unowned references do not impact the reference count of an object. But unlike a weak reference, unowned references are never optional. This means if you try to access an unowned property that points to an object that has already been deallocated, it is like forcefully unwrapping an optional value that is nil. It is important that an unowned reference should only be used if the reference and the object are sure to get deallocated at the same time.

Closures are another way of ending up in reference cycles leading to memory leaks.

For example if a have a property with count my cities

lazy var cityCount: () -> Int { self.city.count }

If I use this property in runApp, my object refers to the closure through the computed property cityCount and the closure refers to the object through self. Since this is a strong reference both ways, it creates a retain cycle.

I have to use Capture Lists to capture a weak or unowned reference to self   
lazy var cityCount: () -> Int { [weak self] in self?.city.count}

The closure does not impact the reference count and as soon as my class goes out of scope, so does the closure.

MRC Manual Reference Counting.

When we use Objective c, we should count references manually.

You should manually make and count references on Objective C by using alloc, new, cioy, mutalbeCopy, retain.. etc.

Since ARC supports everything, it looks like we do not need to care about memory management. However, there is an exception to deal with and It is a retain cycle.

MRC vs Garbage collection

It works in a way such that the runtime detects unused objects and object graphs in the background. This happens at intermediate intervals, either after a certain amount of time has passed or when the runtime memory gets low, and not released at that exact moment. The GC removes the burden of the developer from freeing or destroying the objects explicitly.

GC can clean up the object graphs including retain cycles. All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects. When objects are no longer referenced directly or indirectly by a GC root, they will be removed.

GC needs five times the memory to perform as fast as explicit memory management. If the memory is compromised, it leads to possible stalls in program execution.