

In one line: ARC = the compiler inserts retain/release; an object is freed *deterministically* the instant its **strong** count hits 0 (dealloc runs). No tracing GC ⇒ **no cycle detection**: two objects that strongly own each other leak forever, silently. You break cycles by making the back-edge weak or unowned.

How it works

- ARC manages **class instances** (+ actors, closure contexts). Structs/enums are copied, not counted — but copying a struct that holds a class ref retains that object.
- strong** (default) — owns: +1. **weak** — non-owning, must be var + Optional; the runtime (side table) sets it to nil when the object dies. **unowned** — non-owning, not auto-nilled; access after death = **crash** (unowned(unsafe) = dangling pointer, UB).
- Ownership ≠ nil-ability**. Two independent dials: ? answers “can it be nil?”; strong/weak/unowned answers “does it keep the object alive?”. var x: Foo? retains exactly as hard as var x: Foo.
- weak vs unowned**: can the referent die while I still hold the ref? *Maybe* → weak (delegates, async callbacks — the safe default). *Never* — it outlives me → unowned (child that cannot exist without its parent: in CreditCard, unowned let customer: Customer).
- Cycle shapes**: (1) **delegate** — make the protocol: AnyObject and the property weak var delegate; (2) **closure** stored on self that captures self (completion handlers, lazy var, Combine sink); (3) **parent/child back-refs** — parent owns children strongly, child.parent, list.prev are weak; (4) **Timer** — the run loop retains it: also invalidate().
- Capture list [x]** = a **by-value snapshot taken when the closure is created**. Without a list the closure captures the *variable* (sees later writes). For a **reference type** the snapshot copies the *pointer*: same object, so later mutations of the object are visible. [weak self] is the same snapshot, just non-owning. Mix: [weak self, count].
- Only **escaping** closures stored (directly or transitively) on self need [weak self]; map/forEach are non-escaping.

Example — break the cycle, prove it

```
final class VM {
    var onChange: (() -> Void)? // VM owns the closure
    func bind() {
        onChange = { [weak self] in // closure does NOT own VM
            guard let self else { return } // unwrap once
            self.reload() }
    }
    func reload() {}
}

func test_VM_deallocates() {
    var sut: VM? = VM(); sut?.bind()
    weak var weakSut = sut // observer, not an owner
    sut = nil // drop the last strong ref
    XCTAssertNil(weakSut, "VM leaked - retain cycle?")
}
```

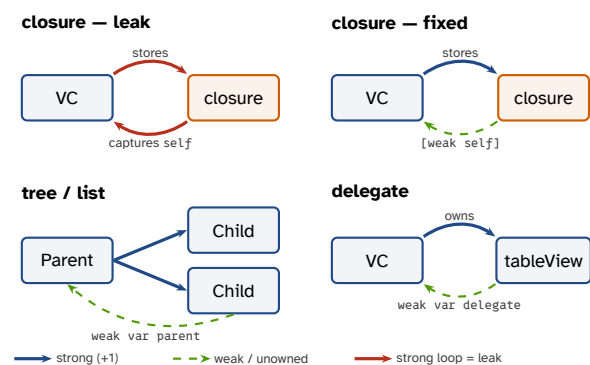
Memory tools — name them

- Xcode Memory Graph Debugger** (debug-bar button): live objects + *who holds whom*; purple ! = leak/cycle. Turn on *Malloc Stack Logging* to see where it was allocated.
- Instruments — Leaks**: finds *unreachable* blocks (true leaks, cycles).
- Instruments — Allocations**: heap growth over time; *Mark Generation* exposes *abandoned* memory (reachable but never freed, e.g. an unbounded cache).
- Zombies**: use-after-free / EXC_BAD_ACCESS.

Picture — the two dials

	Foo (non-optional)	Foo? (optional)	
strong	owns (+1) never nil	still owns (+1) may be nil — not weak!	
weak	compile error (weak must be ?)	non-owning auto-nil on dealloc	? = can be nil
unowned	non-owning crash if dead	non-owning (Swift 5+) not auto-nilled	keyword = keeps alive?

Picture — cycles and their fix



Interview traps

- ! “Optional means non-owning” — your playground mistake. var parent: Node? is *strong*; write weak.
- ! [x] is **by value**, not by reference (Q12): snapshot at creation. Reference type → the *pointer* is copied.
- ! Dealloc test (Q17): weak ref → set strong to nil → XCTAssertNil. A dealloc print is only the manual version.
- ! Tools (Q18, you did not know): Memory Graph, Leaks, Allocations.
- ! “No leaks” but memory grows → abandoned memory → Allocations.
- ! [unowned self] in a network callback that outlives the VC = crash.
- ! Timer + [weak self]: the run loop still holds the timer — invalidate().

Remember

“?” asks *may it be nil*; the keyword asks *do I keep it alive*. Arrows point *down* the tree strong, *back up* weak. Tools: Graph, Leaks, Allocations — “GLA”.

Likely questions

- ARC vs GC? — compile-time counting, deterministic, no pauses, no cycle collection.
- When unowned? — referent provably outlives the reference.
- var n=1; f={[n] in print(n)}; n=99; f()? — prints 1.
- Why weak must be var + ? — ARC writes nil into it.
- Does self.x in a closure capture self? — yes, strongly.