

In one line: **Exclusivity** is the rule (no overlapping access where one is a *write*); **ownership** (borrowing/consuming/~Copyable) is the vocabulary for who owns a value and when it dies; **layout** (MemoryLayout, existentials, refcount headers) is what all of it costs in bytes and atomic ops.

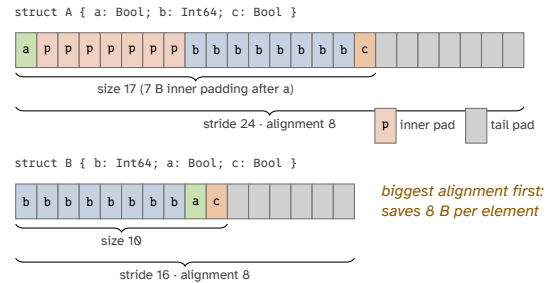
How it works

- **Law of Exclusivity:** two accesses to the same storage may not overlap in time if at least one *writes*. Reads overlap freely. A mutating call or inout argument is *one* write access lasting the **whole call**.
- **Static** check (local vars, provable aliasing) = compile error, zero cost. **Dynamic** check (class properties, globals, vars captured by escaping closures) = runtime begin/end markers, trap “Simultaneous accesses”. On in **Debug and Release** since Swift 5; `-enforce-exclusivity=unchecked` turns the trap into UB. It is *not* a thread-race detector.
- **~Copyable** (5.9) *suppresses* the implicit Copyable — not a protocol. Structs/enums only (not classes or actors). Values are *moved* or *borrowed*, never copied $\Rightarrow$ single owner $\Rightarrow$ a **struct can have deinit**, run exactly once when the owner ends or is consumed.
- **Parameter ownership:** borrowing = +0 / guaranteed, read-only, caller keeps it (default for most params); consuming = +1 / owned, caller's binding is dead after; inout = exclusive write, handed back. `consume x` ends x's lifetime explicitly; `consume func close()` consumes `self`; `discard self` ends it *without* deinit.
- Generics: `<T: ~Copyable>` widens (6.0). Optional, `UnsafePointer` accept noncopyables; `Array` still requires Copyable.
- **MemoryLayout<T>**: `size` = bytes used; `alignment` = start boundary; `stride` = size rounded up to `alignment` = distance between array elements. Swift keeps **declaration order**, so field order sets padding.
- **Existential any P** = 5 words (40 B): 3-word inline buffer + type metadata + 1 witness table per protocol (any P & Q = 48 B; Any = 32 B; class-bound = ref + PWTs). Value > 3 words $\rightarrow$ **boxed on the heap**, buffer holds the pointer. Generics / some P specialise and skip the box.
- **ARC header** = 16 B: metadata (isa) + one 64-bit **inline** refcount word (strong + unowned counts + flags). A weak ref or overflow moves counts to a **side table**; the word then points to it. `swift_retain/release` are **atomic** RMWs — cheap once, costly as traffic, worse under contention. `borrowing/+0` and the optimiser remove pairs.
- **Unmanaged<T>**: manual ARC across a C `void*` context. `passRetained (+1)  $\leftrightarrow$  takeRetainedValue (-1)`; `passUnretained  $\leftrightarrow$  takeUnretainedValue (+0)`. `toOpaque/fromOpaque`.

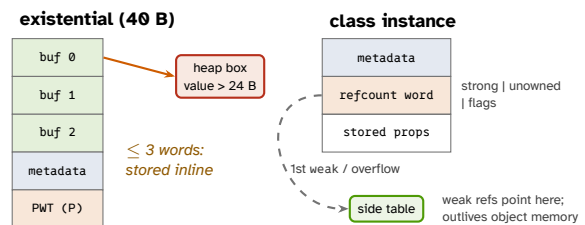
Example

```
struct FileHandle: ~Copyable {
    let fd: Int32
    deinit { close(fd) } // runs exactly once
    consuming func detach() -> Int32 {
        let f = fd; discard self; return f } // no deinit
}
func read(_ h: borrowing FileHandle) {} // lend, +0
func store(_ h: consuming FileHandle) {} // take, +1
let h = FileHandle(fd: 3)
read(h); store(h)
// read(h) error: 'h' used after consume
var x = 1
swap(&x, &x) // error: overlapping accesses to 'x'
```

Picture — padding: same fields, new order



Picture — any P and the object header



Interview traps

- ! Array indexing / pointer maths uses **stride**, not size (Int?: size 9, stride 16).
- ! Exclusivity $\neq$ data-race safety — that is actors / Sendable. It catches single-threaded reentrancy.
- ! `&a[0]`, `&a[1]` conflict: subscript inout accesses the *whole array*. Use `swapAt`.
- ! ~Copyable is suppression, not conformance; a borrowing param cannot be returned or stored (escapes the borrow).
- ! any P in a hot loop may `malloc` (box) and adds dynamic dispatch — prefer some P / generics.
- ! `passRetained + takeUnretainedValue` = leak; the reverse = over-release crash.
- ! “Value type = no ARC” — a struct holding a class ref retains it on copy.

Remember

“**One writer, whole call.**” Borrow reads, Consume takes, Inout rents — “BCI”. Pack fields **big-to-small**. **3 words** fit in an existential; **weak** buys a side table.

Likely questions

1. Why can a ~Copyable struct have deinit? — unique owner, so its end is well-defined.
2. When is exclusivity checked at runtime? — class props, globals, escaping-captured vars.
3. Size of any P on 64-bit? — 40 B (3 buffer + metadata + PWT).
4. Where does the refcount live? — inline in the header; side table once weak-referenced.
5. What does Unmanaged solve? — ARC across a C `void*` callback round-trip.
6. Why is a mutating reentrant callback a trap? — the first exclusive access to `self` is still open.