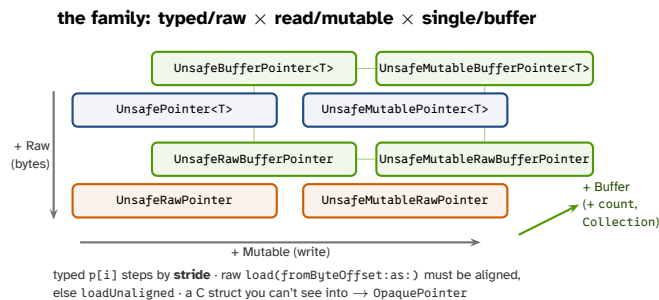
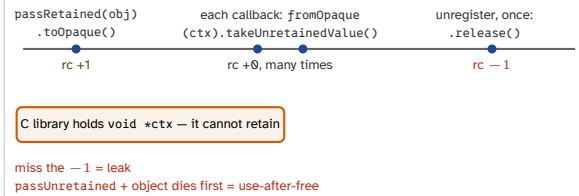


In one line: **Unsafe** = you, not the compiler, own lifetime, initialisation, alignment, type binding and bounds. **Reflection** is deliberately thin: metatypes are values (`T.Type`), **Mirror** only *reads* stored properties, casts query runtime metadata.



Unmanaged: a Swift object through a C void *



How it works — pointers

- States:** allocate → initialize(to:) → use → deinitialize(count:) (releases refs) → deallocate(). Skipping deinitialise on class refs = leak.
- Scoped:** withUnsafePointer(to:), withUnsafeBytes(of:), withUnsafeBufferPointer — the pointer is valid **only inside the closure** (the value may move, COW-copy or die). Implicit `&x` / array / String to a C param: valid for *that call*; keeping one draws a “dangling pointer” warning (5.2+).
- Binding:** memory is bound to one type at a time; typed access through an unrelated type = strict-aliasing UB. `bindMemory(to:capacity:)` rebinds; `withMemoryRebound` rebinds for a scope (e.g. `sockaddr_in` → `sockaddr`); `assumingMemoryBound(to:)` only *asserts*, unchecked. For reinterpretation, prefer raw + `load(as:)`.
- No bounds checks on `p[i]` / `buf[i]` in release. C fixed arrays import as **tuples**; `withExtendedLifetime(obj)` stops ARC releasing an object whose pointer C still uses.
- Unmanaged<T>:** manual retain counts across void *. CF naming rule: *Create/Copy* returns +1 → `takeRetainedValue()`; *Get* → `takeUnretainedValue()`.

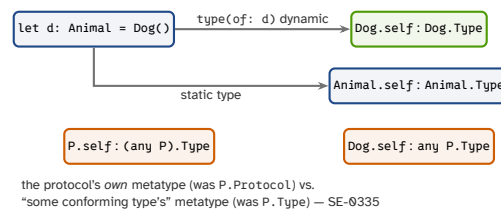
Example — C callback with context

```
final class Counter { var hits = 0 }
let ctx = Unmanaged.passRetained(Counter()).toOpaque() // +1
register_cb({ raw in // @convention(c): no captures
    let c = Unmanaged<Counter>.fromOpaque(raw!)
    .takeUnretainedValue() // +0 borrow
    c.hits += 1 }, ctx)
// later, exactly once:
Unmanaged<Counter>.fromOpaque(ctx).release() // -1
var n: UInt32 = 0xDEADBEEF
let b = withUnsafeBytes(of: &n) { Array($0) } // EF BE AD DE
```

When to reach for it

C/C++ interop, zero-copy binary parsing, a *measured* hot loop. Keep it small, audited, behind a safe API. Swift 6.2's `Span`/`RawSpan` are bounds-checked, non-escaping views that replace many withUnsafe... uses.

Picture — metatypes



How it works — reflection & casts

- `T.self` is the *value* of `T.Type`: call static members and required `init` through it (`decode(User.self, from:)`). `AnyClass` = `AnyObject.Type`; `ObjectIdentifier(T.self)` keys by type. some `P` = one hidden type, static; any `P` = box.
- `Mirror(reflecting:)`: children (label, value), `displayStyle`, `superclassMirror` (inherited props). Stored properties only — **no methods, computed props, setting or construction**. `Slow`; `dump()` uses it.
- `is/as?`: classes walk the superclass chain; protocols look up conformance records (cached, first lookup slow). Casts see through `Optional` inside `Any`; `Int` in `Any` is *not* a `Double`; `as?` any `Sequence` works since 5.7, any `Collection<Int>` needs the iOS 16 runtime.

Interview traps

- ! `withUnsafePointer(to: &x) { $0 }` — returns a dangling pointer. Do the work inside, or copy out.
- ! `size` instead of `stride` for pointer maths; `withUnsafeBytes` is **host** (little-endian) order — use `.bigEndian` for the wire.
- ! `x as?` `AnyObject` always succeeds (boxes a struct). `is Dog` includes subclasses; comparing `type(of: x)` to `Dog.self` is exact.
- ! `Mirror` for serialisation or setting — use `Codable`, key paths.

Remember

“Typed? Mutable? Buffer? — then: in scope, initialised, bound, aligned, in bounds.” Metatype is a value; `Mirror` only reads.

Likely questions

- Pass `self` to a C callback? — `passRetained`, balance once.
- `T.self` vs `type(of:)`? — static vs dynamic metatype.
- Can `Mirror` set a property? — no; read-only, stored props only.