Logical Objects

Universal across programming languages, we have this concept of a value, which is just some amount of fixed data. A value might be the int 5, or a pair of the bool true and the int -20, or an NSRect with the component values (0, 0, 400, 600), or whatever.

In imperative languages we have this concept of an object. It's an unfortunately overloaded term; here I'm using it like the standards do, which is to say that it's a thing that holds a value, but which can be altered at any time to hold a different value. It's tempting to use the word variable instead, and a variable is indeed an object, but "variable" implies all this extra stuff, like being its own independent, self-contained thing, whereas we want a word that also covers fields and array elements and what-have-you. So let's just suck it up and go with "object".

You might also call these "r-value" and "l-value". These have their own connotations that I don't want to get into. Stick with "value" and "object".

In C and C++, every object is physical. It's actually a place in memory somewhere. It's not necessarily easily addressable (because of bitfields), and its lifetime may be tightly constrained (because of temporaries or deallocation), but it's always memory.

In Objective-C, properties and subscripting add an idea of a logical object. The only way you can manipulate it is by calling a function (with unrestricted extra arguments) to either fetch the current value or update it with a new value. The logical object doesn't promise to behave like a physical object, either: for example, you can set it, then immediately get it, and the result might not match the value you set.

Swift has logical objects as well. We have them in a few new places (global objects can be logical), and sometimes we treat objects that are really physical as logical (because resilience prevents us from assuming physicality), and we're considering making some restrictions on how different a logical object can be from a physical object (although set-and-get would still be opaque), but otherwise they're pretty much just like they are in Objective-C.

That said, they do interact with some other language features in interesting ways.

For example, methods on value types have a this parameter. Usually parameters are values, but this is actually an object: if I call a method on an object, and the method modifies the value of this, I expect it to modify the object I called the method on. This is the high-level perspective of what [inout] really means: that what we're really passing as a parameter is an object, not a value. With one exception, everything that follows applies to any sort of [inout] parameter, not just this on value types. More on that exception later.

How do you actually pass an object, though, given that even physical objects might not be addressable, but especially given that an object might be logical?

Well, we can always treat a physical object like a logical object. It's possible to come up with ways to implement passing a logical object (pass a pointer to a struct, the first value of which is a getter, the second value of which is a setter, and the rest of which is opaque to the callee; the struct must be passed to the getter and setter functions). Unfortunately, the performance implications would be terrible: accessing multiple fields would involve multiple calls to the getter, each returning tons of extra information. And getter and setter calls might be very expensive.

We could pass a hybrid format, using direct accesses when possible and a getter/setter when not. Unfortunately, that's a lot of code bloat in every single method implementation.

Or we can always pass a physical, addressable object. This avoids penalizing the fast case where the object is really physical, which is great. For the case where the object is logical, we just have to make it physical somehow. That means materialization: calling the getter, storing the result into temporary memory, passing the temporary, and then calling the setter with the new value in the temporary when the method call is done. This last step is called writeback.

(About that one exception to this all applying equally to [inout]: in addition to all this stuff about calling methods on different kinds of object, we also want to support calling a method on a value. This is also implemented with a form of materialization, which looks just like the logical-object kind except without writeback, because there's nothing to write back to. This is a special rule that only applies to passing this, because we assume that most types will have lots of useful methods that don't involve writing to this, whereas we assume that a function with an explicit [inout] parameter is almost certain to want to write to it.)