orphan:

Summary

Pointer arguments are a fact of life in C and Cocoa, and there's no way we're going to realistically annotate or wrap every API safely. However, there are plenty of well-behaved APIs that use pointer arguments in well-behaved ways that map naturally to Swift argument conventions, and we should interact with those APIs in a natural, Swift-ish way. To do so, I propose adding language and library facilities that enable the following uses of pointer arguments:

- Const pointer arguments `const int *`, including const pointers to ObjC classes `NSFoo * const *`, can be used as "in" array arguments, as `inout` scalar arguments, or as `UnsafeMutablePointer` arguments.
- Non-const pointer arguments to C types, `int *`, can be used as `inout` array or scalar arguments, or as `UnsafeMutablePointer` arguments.
- Non-const pointer arguments to ObjC class types, `NSFoo **`, can be used as `inout` scalar arguments or passed `nil`. (They cannot be used as array arguments or as `UnsafeMutablePointer` arguments.)
- `const void *` and `void *` pointers can be used in the same ways as pointers to any C type (but not ObjC types).

This model intentionally doesn't try to provide a mapping to every possible use case for pointers in C. It also intentionally avoids addressing special cases we could potentially provide higher-level support for. Some particular issues this proposal specifically does not address:

- Pointer return values
- Special handling of `char*` and/or `const char*` arguments
- Special handling of Core Foundation types
- Special handling of `NSError**` arguments
- Precise lifetime of values (beyond the minimal lifetime extension to the duration of a call)
- Overriding of ObjC methods that take pointer arguments with subclass methods that take non-pointer arguments

Design Considerations

Const Pointer Arguments

Arrays

Const pointer arguments are frequently used in both C and Objective-C to take an array of arguments effectively by value. To support this use case, we should support passing a Swift `Array` value to a const pointer argument. An example from Core Foundation is `CGColorCreate`, which takes a `CGFloat` array of color space-specific components:

```
let rgb = CGColorSpaceCreateCalibratedRGB()
let cyan = CGColorCreate(rgb, [0, 1, 1])
```

We are willing to assume that the API is well-behaved and does not mutate the pointed-to array, so we can safely pass an interior pointer to the array storage without worrying about mutation. We only guarantee the lifetime of the array for the duration of the call, so it could potentially be promoted to a stack allocation.

"In" Arguments

Const pointer arguments are also used in many cases where a value is unmodified, but its identity is important. Somewhat more rarely, const pointer arguments are used as pure "in" value arguments with no regard for identity; this is particularly prevalent on platforms like Win32 where there has historically been no standard ABI for passing structs by value, but pure "in" pointer parameters are rare on our platforms. The potential consequences of disregarding value identity with C APIs are too high to allow scalar arguments to implicitly be used as pointer arguments:

```
// From C: void foo(const pthread_mutex_t *);
import Foo

let mutex = pthread_mutex_create()
// This would pass a different temporary buffer on each call--not what you
// want for a mutex!
foo(mutex)
foo(mutex)
foo(mutex)
```

Although const pointers should never be used for actual mutation, we propose that only `inout` scalar arguments be accepted for const pointer parameters. Although our semantics normally do not guarantee value identity, `inout` parameters that refer to stored variables or stored properties of C-derived types are in practice never subjected to implicit writebacks except in limited circumstances such as capture of `inout` references in closures that could be diagnosed. Requiring `inout` also prevents the use of rvalues or `let` bindings that never have well-defined addresses as pointer arguments. This more clearly communicates the intent for the callee to receive the same variable on every call:

```
// From C: void foo(const pthread_mutex_t *);
import Foo
```

```
var mutex = pthread_mutex_create()
foo(&mutex)
foo(&mutex)
foo(&mutex)
```

If using an rvalue as a pointer argument is desired, it can easily be wrapped in an array. This communicates that the value *is* being copied into the temporary array, so it's more obvious that identity would not be maintained:

```
// an immutable scalar we might want to pass into a "const double*".
let grayLevel = 0.5
let monochrome = CGColorSpaceCreateGrayscale()

// error, can't pass Double into second argument.
let c1 = CGColorCreate(monochrome, grayval)
// error, can't take the address of a 'let' (would be ok for a 'var')
let c2 = CGColorCreate(monochrome, &grayval)
// OK, we're explicitly forming an array
let c3 = CGColorCreate(monochrome, [grayval])
```

Non-Const Pointer Arguments

C Types

Non-const arguments of C type can be used as "out" or "inout" parameters, either of scalars or of arrays, and so should accept `inout` parameters of array or scalar type. Although a C API may expect a pure "out" parameter and not require initialization of its arguments, it is safer to assume the argument is `inout` and always require initialization:

```
var s, c: Double
// error, 's' and 'c' aren't initialized
sincos(0.5, &s, &c)

var s1 = 0.0, c1 = 0.0
// OK
sincos(0.5, &s1, &c1)
```

For array parameters, the exact point of mutation inside the callee cannot be known, so a copy-on-write array buffer must be eagerly uniqued prior to the address of the array being taken:

```
func loadFloatsFromData(_ data: NSData) {
  var a: [Float] = [0.0, 0.0, 0.0, 0.0]
  var b = a

  // Should only mutate 'b' without affecting 'a', so its backing store
  // must be uniqued
  data.getBytes(&b, sizeof(Float.self) * b.count)
}
```

ObjC Types

ARC semantics treat an `NSFoo**` type as a pointer to an `__autoreleasing NSFoo*`. Although in theory these interfaces could receive arrays of object pointers in Objective-C, that use case doesn't come up in Cocoa, and we can't reliably bridge such APIs into Swift. We only need to bridge ObjC mutable pointer types to accept a scalar `inout` object reference or `nil`.

Pointer Return Values

This proposal does not address the handling of return values, which should still be imported into Swift as `UnsafeMutablePointer` values.

Library Features

The necessary conversions can be represented entirely in the standard library with the help of some new language features, inout address conversion, inout writeback conversion, and interior pointer conversion, described below. There are three categories of argument behavior needed, and thus three new types. These types should have no user-accessible operations of their own other than their implicit conversions. The necessary types are as follows:

- `CConstPointer<T>` is the imported representation of a `const T *` argument. It is implicitly convertible from `inout T` by inout address conversion and from `Array<T>` by immutable interior pointer conversion. It is also implicitly convertible to and from `UnsafeMutablePointer<T>` by normal conversion.
- `CMutablePointer<T>` is the imported representation of a `T *` argument for a POD C type `T`. It is implicitly convertible from `inout T` by inout address conversion and from `inout Array<T>` by mutating interior pointer conversion. It is also implicitly convertible to and from `UnsafeMutablePointer<T>` by normal conversion.
- `ObjCInOut<T>` is the imported representation of a `T **` argument for an ObjC class type `T`. It is implicitly convertible from `inout T` by inout writeback conversion and is implicitly convertible from `nil`. It cannot be converted from an array or to `UnsafeMutablePointer`.

New Language Features

To support the necessary semantics for argument passing, some new conversion forms need to be supported by the language with special-cased lifetime behavior.

Interior Pointer Conversions

To be able to pass a pointer to array data as an argument, we need to be able to guarantee the lifetime of the array buffer for the duration of the call. If mutation can potentially occur through the pointer, then copy-on-write buffers must also be uniqued prior to taking the address. A new form of conversion, `@unsafe_interior_pointer_conversion`, can be applied to an instance method of a type, to allow that type to return both a converted pointer and an owning reference that guarantees the validity of the pointer. Such methods can be either `mutating` or non-mutating; only non-mutating conversions are considered for non- `inout` parameters, and only `mutating` conversions are considered for `inout` parameters:

```
extension Array {
  @unsafe_interior_pointer_conversion
  func convertToConstPointer()
  -> (CConstPointer<T>, ArrayBuffer<T>) {
    return (CConstPointer(self.base), self.owner)
  }

  @unsafe_interior_pointer_conversion
  mutating func convertToMutablePointer()
  -> (CMutablePointer<T>, ArrayBuffer<T>) {
    // Make the backing buffer unique before handing out a mutable pointer.
    self.makeUnique()
    return (CMutablePointer(self.base), self.owner)
  }
}
```

`@unsafe_interior_pointer_conversion` conversions are only considered in argument contexts. If such a conversion is found, the first element of the return tuple is used as the argument, and a strong reference to the second element is held for the duration of the callee that receives the converted argument.

Inout Address Conversion

To pass an `inout` as a pointer argument, we need to be able to lock an address for the `inout` for the duration of the call, which is not normally possible. This functionality only needs to be available to the standard library, so can be expressed in terms of builtins. A type can conform to the `_BuiltinInOutAddressConvertible` protocol to be convertible from an inout reference. The protocol is defined as follows:

```
protocol _BuiltinInOutAddressConvertible {
  /// The type from which inout conversions are allowed to the conforming
  /// type.
  typealias InOutType

  /// Create a value of the conforming type using the address of an inout
  /// argument.
  class func _convertFromInOutAddress(_ p: Builtin.RawPointer) -> Self
}
```

An example of a conformance for `CMutablePointer`:

```
struct CMutablePointer<T>: _BuiltinInOutAddressConvertible {
  let ptr: Builtin.RawPointer

  typealias InOutType = T

  @_transparent
  static func _convertFromInOutAddress(_ p: Builtin.RawPointer)
  -> CMutablePointer {
    return CMutablePointer(p)
  }
}

func foo(_ p: CMutablePointer<Int>) { }

var i = 0
foo(&i)
```

The lifetime of the variable, stored property owning object, or writeback buffer backing the inout is guaranteed for the lifetime of the callee that receives the converted parameter, as if the callee had received the inout parameter directly.

Inout Writeback Conversion

Inout address conversion alone is not enough for `ObjCInOut` to work as intended, because the change to the `__autoreleasing`

convention for the pointed-to object reference requires a writeback temporary. The `_BuiltinInOutWritebackConvertible` protocol allows for an additional writeback to be introduced before and after the address of the `inout` is taken:

```
protocol _BuiltinInOutWritebackConvertible {
  /// The original type from which inout conversions are allowed to the
  /// conforming type.
  typealias InOutType

  /// The type of the temporary writeback whose address is used to construct
  /// the converted value.
  typealias WritebackType

  /// Get the initial value the writeback temporary should have on entry to
  /// the call.
  class func _createWriteback(inout InOutType) -> WritebackType

  /// Create a value of the conforming type using the address of the writeback
  /// temporary.
  class func _convertFromWritebackAddress(_ p: Builtin.RawPointer) -> Self

  /// Write the writeback temporary back to the original value.
  class func _commitWriteback(inout InOutType, WritebackType)
}
```

An example of a conformance for `ObjCInOut`:

```
struct ObjCInOut<T: class>: _BuiltinInOutWritebackConvertible {
  let ptr: Builtin.RawPointer

  typealias InOutType = T!
  typealias WritebackType = Builtin.RawPointer

  @_transparent
  static func _createWriteback(ref: inout T!)
  -> Builtin.RawPointer {
    // The initial object reference is passed into the callee effectively
    // __unsafe_unretained, so pass it as a RawPointer.
    return unsafeBitCast(ref, Builtin.RawPointer.self)
  }

  @_transparent
  static func _commitWriteback(ref: inout T!,
                               value: Builtin.RawPointer) {
    // The reference is autoreleased on return from the caller, so retain it
    // by loading it back as a T?.
    ref = unsafeBitCast(value, T!.self)
  }

  @_transparent
  static func _convertFromWritebackAddress(_ value: Builtin.RawPointer) {
    return ObjCInOut(value)
  }
}
```

The lifetime of the writeback is guaranteed for the lifetime of the callee that receives the converted parameter, as if the callee had received the writeback temporary as a mutable logical property of the original inout parameter.