Swift Performance Predictability

Joe Groff's performance roadmap forum pitch.

https://forums.swift.org/t/a-roadmap-for-improving-swift-performance-predictability-arc-improvements-and-ownership-control/54206

A Flock of Swifts discussion 1/8/2021

Roadmap

- Pitch for upcoming changes being introduced by Apple’s performance team.
- Changes aimed to make automatic reference count (ARC) model easier to understand and control from a high-level.
- Reduce ARC traffic 🚦🚗🚙🚗
- Progressive disclosure - those dealing with non-performance sensitive code can ignore these features for the most part.
- Expect many of these changes to appear in 2022. (Pitches in the next few months)

Lexical Lifetime

```
class Controller {
  weak var delegate: MyDelegate?

  func callDelegate() {
    _ = delegate!
  }
}

let delegate = MyDelegate(controller)
MyController(delegate).callDelegate()
```

- Lexical lifetime make object lifetimes easier to reason about
- Lessens need for withExtendedLifetime
- Ordering not as strict as C++

move function for explicit ownership transfer

```
struct SortedArray {
    var values: [String]

    init(values: [String]) {
        self.values = values
        // Ensure the values are actually sorted
        self.values.sort()
    }
}
```

```
struct SortedArray {
    var values: [String]

    init(values: [String]) {
        // Ensure that, if `values` is uniquely referenced, it remains so,
        // by moving it into `self`
        self.values = move(values)
        // Ensure the values are actually sorted
        self.values.sort()
    }
}
```

Managing ownership transfer across calls with argument modifiers

```
extension Array {
    mutating func append(_ value: consuming Element) { ... }
}
```

```
struct Foo {
    var bars: [Bar]

    // `name` is only used for logging, so making it `nonconsuming`
    // saves a retain on the caller side
    init(bars: [Bar], name: nonconsuming String) {
        print("creating Foo with name \(name)")
        self.bars = move(bars)
    }
}
```

- Consuming and non-consuming
- Today init, setters parameters are consuming
- Regular function parameters, inout are non-consuming
- Replaces __owned and __shared

read and modify accessor coroutines for in-place borrowing and mutation of data structures

```
struct Foo {
    private var _x: [Int]

    var x: [Int] {
        get { return _x }
        set { _x = newValue }
    }
}
```

```
foo.x.append(1738)
```

```
var foo_x = foo.get_x()
foo_x.append(1738)
foo.set_x(foo_x)
```

```
struct Foo {
    private var _x: [Int]

    var x: [Int] {
        read { yield _x }
        modify { yield &_x }
    }
}
```

- Setters and getter overhead, very expensive for copy on write types
- Standard library currently uses *single-yield* coroutines _read and _modify in Array, Dictionary and Set.

Requiring explicit copies on variables

```swift
class C {}

func borrowTwice(first: C, second: C) {}
func consumeTwice(first: consuming C, second: consuming C) {}
func borrowAndModify(first: C, second: inout C) {}

func foo(x: @noImplicitCopy C) {
    // This is fine. We can borrow the same value to use it as a
    // nonconsuming argument multiple times.
    borrowTwice(first: x, second: x)

    // This would normally require copying x twice, because
    // `consumeTwice` wants to consume both of its arguments, and
    // we want x to remain alive for use here too.
    // @noImplicitCopy would flag both of these call sites as needing
    // `copy`.
    consumeTwice(first: x, second: x) // error: copies x, which is marked noImplicitCopy
    consumeTwice(first: copy(x), second: copy(x)) // OK

    // This would also normally require copying x once, because
    // modifying x in-place requires exclusive access to x, so
    // the `first` immutable argument would receive a copy instead
    // of a borrow to avoid breaking exclusivity.
    borrowAndModify(first: copy(x), second: &x)

    // Here, we can `move` the second argument, since it is the final
    // use of `x`
    consumeTwice(first: copy(x), second: move(x))
}
```

Generalized nonescaping arguments

```
func withUnsafePointer<T, R>(to: T, _ body: (@nonescaping UnsafePointer<T>)
-> R) -> R

let x = 42
var xp: UnsafePointer<Int>? = nil
withUnsafePointer(to: x) { p in
    xp = p // error! can't escape p
}
```

- Like what we have for closures
- Makes the optimizer more powerful
- Can prevent programmer errors with unsafe code

Borrow variables

```
ref greatAunt = mother.father.sister
greatAunt.sayHello()
mother.father.sister = otherGreatAunt // error, 
can't mutate `mother.father.sister` while 
`greatAunt` borrows it
greatAunt.sayGoodbye()
```

- Reaching deep into a hierarchy, multiple accesses are reasons to want borrow or inout binding

- The ref is borrowed from C# but not a perfect name since it might not actually be a reference.

Looking forward to move-only types

- Uniquely owned types which have no ARC overhead
- Take no-implicit-copy and non escaping to the type-level
- Requires big changes to the generics model so not included here
- The proposed changes clear a path for move-only types