Standard Library Programmers Manual

This is meant to be a guide to people working on the standard library. It covers coding standards, code organization, best practices, internal annotations, and provides a guide to standard library internals. This document is inspired by LLVM's excellent programmer's manual and coding standards.

TODO: Should this subsume or link to StdlibRationales.rst?

TODO: Should this subsume or link to AccessControlInStdlib.rst

In this document, "stdlib" refers to the core standard library (`stdlib/public/core`), our Swift overlays for system frameworks (`stdlib/public/Darwin/*` , `stdlib/public/Windows/*` , etc.), as well as the auxiliary and prototype libraries under `stdlib/private` .

Coding style

Formatting Conventions

The Standard Library codebase has some uniformly applied formatting conventions. While these aren't currently automatically enforced, we still expect these conventions to be followed in every PR, including draft PRs. (PRs are first and foremost intended to be read/reviewed by people, and it's crucial that trivial formatting issues don't get in the way of understanding proposed changes.)

Some of this code is very subtle, and its presentation matters greatly. Effort spent on getting formatting *just right* is time very well spent: new code we add is going to be repeatedly read and re-read by many people, and it's important that code is presented in a way that helps understanding it.

Line Breaking

The stdlib currently has a hard line length limit of 80 characters. This allows code to be easily read in environments that don't gracefully handle long lines, including (especially!) code reviews on GitHub.

We use two spaces as the unit of indentation. We don't use tabs.

To break long lines, please closely follow the indentation conventions you see in the existing codebase. (FIXME: Describe in detail.)

Our primary rule is that if we need to insert a line break anywhere in the middle of a list (such as arguments, tuple or array/dictionary literals, generic type parameters, etc.), then we must go all the way and put each item on its own line, indented by +1 unit, even if some of the items would fit on a single line together.

The rationale for this is that line breaks tend to put strong visual emphasis on the item that follows them, risking subsequent items on the same line to be glanced over during review. For example, see how easy it is to accidentally miss `arg2` in the second example below.

```
// BAD (completely unreadable)
@inlinable public func foobar<Result>(_ arg1: Result, arg2: Int, _ arg3: (Result,
Element) throws -> Result) rethrows -> Result {
  ...
}

// BAD (arg2 is easily missed)
@inlinable
public func foobar<Result>(
```

```
  _ arg1: Result, arg2: Int,              // ☹
  _ arg3: (Result, Element) throws -> Result
) rethrows -> Result {

// GOOD
@inlinable
public func foobar<Result>(
  _ arg1: Result,
  arg2: Int,
  _ arg3: (Result, Element) throws -> Result
) rethrows -> Result {
  ...
}
```

As a special case, function arguments that are very tightly coupled together are sometimes kept on the same line. The typical example for this is a pair of defaulted file/line arguments that track the caller's source position:

```
// OK
internal func _preconditionFailure(
  _ message: StaticString = StaticString(),
  file: StaticString = #file, line: UInt = #line
) -> Never {
  ...
}

// Also OK
internal func _preconditionFailure(
  _ message: StaticString = StaticString(),
  file: StaticString = #file,
  line: UInt = #line
) -> Never {
  ...
}
```

(When in doubt, err on the side of adding more line breaks.)

For lists that have delimiter characters (`(` / `)` , `[` / `]` , `<` / `>` , etc.), we prefer to put a line break both *after* the opening delimiter, and *before* the closing delimiter. However, within function bodies, it's okay to omit the line break before the closing delimiter.

```
// GOOD:
func foo<S: Sequence, T>(
  input: S,
  transform: (S.Element) -> throws T
) -> [S.Element] {     // Note: there *must* be a line break before the ')'
  ...
  someLongFunctionCall(
    on: S,
    startingAt: i,
    stride: 32)        // Note: the break before the closing paren is optional
}
```

If the entire contents of a list fit on a single line, it is okay to only break at the delimiters. That said, it is also acceptable to put breaks around each item:

```
// GOOD:
@_alwaysEmitIntoClient
internal func _parseIntegerDigits<Result: FixedWidthInteger>(
  ascii codeUnits: UnsafeBufferPointer<UInt8>, radix: Int, isNegative: Bool
) -> Result? {
  ...
}

// ALSO GOOD:
@_alwaysEmitIntoClient
internal func _parseIntegerDigits<Result: FixedWidthInteger>(
  ascii codeUnits: UnsafeBufferPointer<UInt8>,
  radix: Int,
  isNegative: Bool
) -> Result? {
  ...
}
```

The rules typically don't require breaking lines that don't exceed the length limit; but if you find it helps understanding, feel free to do so anyway.

```
// OK
guard let foo = foo else { return false }

// Also OK
guard let foo = foo else {
  return false
}
```

Historically, we had a one (1) exception to the line limit, which is that we allowed string literals to go over the margin. Now that Swift has multi-line string literals, we could start breaking overlong ones. However, multiline literals can be a bit heavy visually, while in most cases the string is a precondition failure message, which doesn't necessarily need to be emphasized as much -- so the old exception still applies:

```
      // OK
      _precondition(                                                    |
        buffer.baseAddress == firstElementAddress,                      |
        "Can't reassign buffer in
Array(unsafeUninitializedCapacity:initializingWith:)"
      )                                                                 |
                                                                        |
      // Also OK, although spending 4 lines on the message is a bit much |
      _precondition(                                                    |
        buffer.baseAddress == firstElementAddress,                      |
        """                                                             |
        Can't reassign buffer in \                                      |
```

```
        Array(unsafeUninitializedCapacity:initializingWith:)                    |
        """                                                                     |
      )                                                                         |
```

In every other case, long lines must be broken up. We expect this rule to be strictly observed.

Presentation of Type Definitions

To ease reading/understanding type declarations, we prefer to define members in the following order:

1. Crucial type aliases and nested types, not exceeding a handful of lines in length
2. Stored properties
3. Initializers
4. Any other instance members (methods, computed properties, etc)

Please keep all stored properties together in a single uninterrupted list, followed immediately by the type's most crucial initializer(s). Put these as close to the top of the type declaration as possible -- we don't want to force readers to scroll around to find these core definitions.

We also have some recommendations for defining other members. These aren't strict rules, as the best way to present definitions varies; but it usually makes sense to break up the implementation into easily digestable, logical chunks.

- In general, it is a good idea to keep the main `struct` / `class` definiton as short as possible: preferably it should consist of the type's stored properties and a handful of critical initializers, and nothing else.
- Everything else should go in standalone extensions, arranged by logical theme. For example, it's often nice to define protocol conformances in dedicated extensions. If it makes sense, feel free to add a comment to title these sectioning extensions.
- Think about what order you present these sections -- put related conformances together, follow some didactic progression, etc. E.g, conformance definitions for closely related protocols such as `Equatable` / `Hashable` / `Comparable` should be kept very close to each other, for easy referencing.
- In some cases, it can also work well to declare the most essential protocol conformances directly on the type definition; feel free to do so if it helps understanding. (You can still implement requirements in separate extensions in this case, or you can do it within the main declaration.)
- It's okay for the core type declaration to forward reference large nested types or static members that are defined in subsequent extensions. It's often a good idea to define these in an extension immediately following the type declaration, but this is not a strict rule.

Extensions are a nice way to break up the implementation into easily digestable chunks, but they aren't the only way. The goal is to make things easy to understand -- if a type is small enough, it may be best to list every member directly in the `struct` / `class` definition, while for huge types it often makes more sense to break them up into a handful of separate source files instead.

```
// BAD (a jumbled mess)
struct Foo: RandomAccessCollection, Hashable {
  var count: Int { ... }

  struct Iterator: IteratorProtocol { /* hundreds of lines */ }

  class _Storage { /* even more lines */ }
```

```
  static func _createStorage(_ foo: Int, _ bar: Double) -> _Storage { ... }

  func hash(into hasher: inout Hasher) { ... }

  func makeIterator() -> Iterator { ... }

  /* more stuff */

  init(foo: Int, bar: Double) {
    _storage = Self._createStorage(foo, bar)
  }

  static func ==(left: Self, right: Self) -> Bool { ... }

  var _storage: _Storage
}

// GOOD
struct Foo {
  var _storage: _Storage

  init(foo: Int, bar: Double) { ... }
}

extension Foo {
  class _Storage { /* even more lines */ }

  static func _createStorage(_ foo: Int, _ bar: Double) -> _Storage { ... }
}

extension Foo: Equatable {
  static func ==(left: Self, right: Self) -> Bool { ... }
}

extension Foo: Hashable {
  func hash(into hasher: inout Hasher) { ... }
}

extension Foo: Sequence {
  struct Iterator: IteratorProtocol { /* hundreds of lines */ }

  func makeIterator() -> Iterator { ... }
  ...
}

extension Foo: RandomAccessCollection {
  var count: Int { ... }
  ...
}

extension Foo {
```

```
  /* more stuff */
}
```

Public APIs

Core Standard Library

All new public API additions to the core Standard Library must go through the Swift Evolution Process. The Core Team must have approved the additions by the time we merge them into the stdlib codebase.

All public APIs should come with documentation comments describing their purpose and behavior. It is highly recommended to use big-oh notation to document any guaranteed performance characteristics. (CPU and/or memory use, number of accesses to some underlying collection, etc.)

Note that implementation details are generally outside the scope of the Swift Evolution -- the stdlib is free to change its internal algorithms, internal APIs and data structures etc. from release to release, as long as the documented API (and ABI) remains intact.

For example, since `hashValue` was always documented to allow changing its return value across different executions of the same program, we were able to switch to randomly seeded hashing in Swift 4.2 without going through the Swift Evolution process. However, the introduction of `hash(into:)` required a formal proposal. (Note though that highly visible behavioral changes like this are much more difficult to implement now that they were in the early days -- in theory we can still do ABI-preserving changes, but Hyrum's Law makes it increasingly more difficult to change any observable behavior. For example, in some cases we may need to add runtime version checks for the Swift SDK on which the main executable was compiled, to prevent breaking binaries compiled with previous releases.)

We sometimes need to expose some internal APIs as `public` for technical reasons (such as to interoperate with other system frameworks, and/or to enable testing/debugging certain functionality). We use the Leading Underscore Rule (see below) to differentiate these from the documented stdlib API. Underscored APIs aren't considered part of the public API surface, and as such they don't need to go through the Swift Evolution Process. Regular Swift code isn't supposed to directly call these; when necessary, we may change their behavior in source-incompatible ways or we may even remove them. (However, such changes are technically ABI breaking, so they need to be carefully considered against the risk of breaking existing binaries.)

Overlays and Private Code

The overlays specific to particular platforms generally have their own API review processes. These are outside the scope of Swift Evolution.

Anything under `stdlib/private` can be added/removed/changed with the simple approval of a stdlib code owner.

The Leading Underscore Rule

All APIs that aren't part of the stdlib's official public API must include at least one underscored component in their fully qualified names. This includes symbols that are technically declared `public` but that aren't considered part of the public stdlib API, as well as `@usableFromInline internal`, plain `internal` and `[file]private` types and members.

The underscored component need not be the last. For example, `Swift.Dictionary._worble()` is a good name for an internal helper method, but so is `Swift._NativeDictionary.worble()` -- the `_NativeDictionary` type already has the underscore.

Initializers don't have a handy base name on which we can put the underscore; instead, we put the underscore on the first argument label, adding one if necessary: e.g., `init(_ value: Int)` may become `init(_value: Int)`. If the initializer doesn't have any parameters, then we add a dummy parameter of type Void with an underscored label: for example, `UnsafeBufferPointer.init(_empty: ())`.

This rule ensures we don't accidentally clutter the public namespace with `@usableFromInline` things (which could prevent us from choosing the best names for newly public API later), and it also makes it easy to see at a glance if a piece of stdlib code uses any non-public entities.

Explicit Access Modifiers

We prefer to explicitly spell out all access modifiers in the stdlib codebase. (I.e., we type `internal` even if it's the implicit default.) Additionally, we put the access level on each individual entry point rather than inheriting them from the extension they are in:

```
public extension String {
  // 😢👎
  func blanch() { ... }
  func roast() { ... }
}

extension String {
  // 😊👍
  public func blanch() { ... }
  public func roast() { ... }
}
```

This makes it trivial to identify the access level of every definition without having to scan the context it appears in.

For historical reasons, the existing codebase generally uses `internal` as the catch-all non-public access level. However, it is okay to use `private` / `fileprivate` when appropriate.

Availability

Every entry point in the standard library that has an ABI impact must be applied an `@available` attribute that describes the earliest ABI-stable OS releases that it can be deployed on. (Currently this only applies to macOS, iOS, watchOS and tvOS, since Swift isn't ABI stable on other platforms yet.)

Just like access control modifiers, we prefer to put `@available` attributes on each individual access point, rather than just the extension in which they are defined.

```
// 😢👎
@available(SwiftStdlib 5.2, *)
extension String {
  public func blanch() { ... }
  public func roast() { ... }
}

//
extension String {
  @available(SwiftStdlib 5.2, *)
  public func blanch() { ... }
```

```
  @available(SwiftStdlib 5.2, *)
  public func roast() { ... }
}
```

This coding style is enforced by the ABI checker -- it will complain if an extension member declaration that needs an availability doesn't have it directly attached.

This repository defines a set of availability macros (of the form `SwiftStdlib x.y`) that map Swift Stdlib releases to the OS versions that shipped them, for all ABI stable platforms. The following two definitions are equivalent, but the second one is less error-prone, so we prefer that:

```
extension String {
  // 🥺🐟👎
  @available(macOS 10.15.4, iOS 13.4, watchOS 6.2, tvOS 13.4, *)
  public func fiddle() { ... }

  // 😎👍
  @available(SwiftStdlib 5.2, *)
  public func fiddle() { ... }
}
```

(Mistakes in the OS version number list are very easy to miss during review, but can have major ABI consequences.)

This is especially important for newly introduced APIs, where the corresponding OS releases may not even be known yet.

Features under development that haven't shipped yet must be marked as available in the placeholder OS version `9999`. This special version is always considered available in custom builds of the Swift toolchain (including development snapshots), but not in any ABI-stable production release.

Never explicitly spell out such placeholder availability -- instead, use the `SwiftStdlib` macro corresponding to the Swift version we're currently working on:

```
// 🥺🐟👎
@available(macOS 9999, iOS 9999, watchOS 9999, tvOS 9999, *)
public struct FutureFeature {
  ...
}

// 😎👍
@available(SwiftStdlib 6.3, *) // Or whatever
public struct FutureFeature {
  ...
}
```

This way, platform owners can easily update declarations to the correct set of version numbers by simply changing the definition of the macro, rather than having to update each individual declaration.

If we haven't defined a version number for the "next" Swift release yet, please use the special placeholder version `SwiftStdlib 9999`, which always expands to 9999 versions. Declarations that use this version will need to be

manually updated once we decide on the corresponding Swift version number.

Internals

Unwrapping Optionals

Optionals can be unwrapped with `!`, which triggers a trap on nil. Alternatively, they can be `.unsafelyUnwrapped()`, which will check and trap in debug builds of user code. Internal to the standard library is `._unsafelyUnwrappedUnchecked()` which will only check and trap in debug builds of the standard library itself. These correspond directly with `_precondition`, `_debugPrecondition`, and `_internalInvariant`. See that section for details.

UnsafeBitCast and Casting References

In general `unsafeBitCast` should be avoided because its correctness relies on subtle assumptions that will never be enforced, and it indicates a bug in Swift's type system that should be fixed. It's less bad for non-pointer trivial types. Pointer casting should go through one of the memory binding API instead as a last resort.

Reference casting is more interesting. References casting can include converting to an Optional reference and converting from a class constrained existential.

The regular `as` operator should be able to convert between reference types with full dynamic checking.

`unsafeDownCast` is just as capable, but is only dynamically checked in debug mode or if the cast requires runtime support.

`_unsafeUncheckedDowncast` is the same but is only dynamically checked in the stdlib asserts build, or if the cast requires runtime support.

`_unsafeReferenceCast` is only dynamically checked if the cast requires runtime support. Additionally, it does not impose any static `AnyObject` constraint on the incoming reference. This is useful in a generic context where the object-ness can be determined dynamically, as done in some bridged containers.

Builtins

`_fastPath` and `_slowPath`

`_fastPath` returns its argument, wrapped in a Builtin.expect. This informs the optimizer that the vast majority of the time, the branch will be taken (i.e. the then branch is “hot”). The SIL optimizer may alter heuristics for anything dominated by the then branch. But the real performance impact comes from the fact that the SIL optimizer will consider anything dominated by the else branch to be infrequently executed (i.e. “cold”). This means that transformations that may increase code size have very conservative heuristics to keep the rarely executed code small.

The probabilities are passed through to LLVM as branch weight metadata, to leverage LLVM’s use of GCC style builtin_expect knowledge (e.g. for code layout and low-level inlining).

`_fastPath` probabilities are compounding, see the example below. For this reason, it can actually degrade performance in non-intuitive ways as it marks all other code (including subsequent `_fastPath`s) as being cold. Consider `_fastPath` as basically spraying the rest of the code with a Mr. Freeze-style ice gun.

`_slowPath` is the same as `_fastPath`, just with the branches swapped.

Example:

```
if _fastPath(...) {
  // 90% of the time we execute this: aggressive inlining
  ...
  return
}
// 10% of the time we execute this: very conservative inlining
...
if _fastPath(...) {
  // 9% of the time we execute this: very conservative inlining
    ...
    return
}

// 1% of the time we execute this: very conservative inlining
...
return
```

`_onFastPath`

This should be rarely used. It informs the SIL optimizer that any code dominated by it should be treated as the innermost loop of a performance critical section of code. It cranks optimizer heuristics to 11. Injudicious use of this will degrade performance and bloat binary size.

`_precondition`, `_debugPrecondition`, and `_internalInvariant`

These three functions are assertions that will trigger a run time trap if violated.

- `_precondition` executes in all build configurations. Use this for invariant enforcement in all user code build configurations
- `_debugPrecondition` will execute when **user code** is built with assertions enabled. Use this for invariant enforcement that's useful while debugging, but might be prohibitively expensive when user code is configured without assertions.
- `_internalInvariant` will execute when **standard library code** is built with assertions enabled. Use this for internal only invariant checks that useful for debugging the standard library itself.

`_fixLifetime`

A call to `_fixLifetime` is considered a use of its argument, meaning that the argument is guaranteed to live at least up until the call. It is otherwise a nop. This is useful for guaranteeing the lifetime of a value while inspecting its physical layout. Without a call to `_fixLifetime`, the last formal use may occur while the value's bits are still being munged.

Example:

```
var x = ...
defer { _fixLifetime(x) } // Guarantee at least lexical lifetime for x
let theBits = unsafeBitCast(&x, ...)
... // use of theBits in ways that may outlive x if it weren't for the _fixLifetime
call
```

Annotations

`@_transparent`

Should only be used if necessary. This has the effect of forcing inlining to occur before any dataflow analyses take place. Unless you specifically need this behavior, use `@_inline(__always)` or some other mechanism. Its primary purpose is to force the compiler's static checks to peer into the body for diagnostic purposes.

Use of this attribute imposes limitations on what can be in the body. For more details, refer to the documentation.

`@unsafe_no_objc_tagged_pointer`

This is currently used in the standard library as an additional annotation applied to @objc protocols signifying that any objects which conform to this protocol are not tagged. This means that (on Darwin platforms) such references, unlike AnyObject, have spare bits available from things like restricted memory spaces or alignment.

`@_silgen_name`

This attribute specifies the name that a declaration will have at link time. It is used for two purposes, the second of which is currently considered bad practice and should be replaced with shims:

1. To specify the symbol name of a Swift function so that it can be called from Swift-aware C. Such functions have bodies.
2. To provide a Swift declaration which really represents a C declaration. Such functions do not have bodies.

Using `@_silgen_name` to call Swift from Swift-aware C

Rather than hard-code Swift mangling into C code, `@_silgen_name` is used to provide a stable and known symbol name for linking. Note that C code still must understand and use the Swift calling convention (available in swift-clang) for such Swift functions (if they use Swift's CC). Example:

```
@_silgen_name("_destroyTLS")
internal func _destroyTLS(_ ptr: UnsafeMutableRawPointer?) {
  // ... implementation ...
}
```

```
SWIFT_CC(swift) SWIFT_RUNTIME_STDLIB_INTERNAL
void _destroyTLS(void *);

// ... C code can now call _destroyTLS on a void * ...
```

Using `@_silgen_name` to call C from Swift

The standard library cannot import the Darwin module (much less an ICU module), yet it needs access to these C functions that it otherwise wouldn't have a decl for. For that, we use shims. But, `@_silgen_name` can also be used on a body-less Swift function declaration to denote that it's an external C function whose symbol will be available at link time, even if not available at compile time. This usage is discouraged.

Internal structures

`_FixedArray16`

The standard library has an internal array type of fixed size 16. This provides fast random access into contiguous (usually stack-allocated) memory. See FixedArray.swift for implementation.

Thread Local Storage

The standard library utilizes thread local storage (TLS) to cache expensive computations or operations in a thread-safe fashion. This is currently used for tracking some ICU state for Strings. Adding new things to this struct is a little more involved, as Swift lacks some of the features required for it to be expressed elegantly (e.g. move-only structs):

1. Add the new member to `_ThreadLocalStorage` and a static `getMyNewMember` method to access it. `getMyNewMember` should be implemented using `getPointer`.
2. If the member is not trivially initializable, update `_initializeThreadLocalStorage` and `_ThreadLocalStorage.init`.
3. If the field is not trivially destructable, update `_destroyTLS` to properly destroy the value.

See ThreadLocalStorage.swift for more details.

Working with Resilience

Maintaining ABI compatibility with previously released versions of the standard library makes things more complicated. This section details some of the extra rules to remember and patterns to use.

The Curiously Recursive Inlinable Switch Pattern (CRISP)

When inlinable code switches over a non-frozen enum, it has to handle possible future cases (since it will be inlined into a module outside the standard library). You can see this in action with the implementation of `round(_:)` in FloatingPointTypes.swift.gyb, which takes a FloatingPointRoundingRule. It looks something like this:

```
@_transparent
public mutating func round(_ rule: FloatingPointRoundingRule) {
  switch rule {
  case .toNearestOrAwayFromZero:
    _value = Builtin.int_round_FPIEEE${bits}(_value)
  case .toNearestOrEven:
    _value = Builtin.int_rint_FPIEEE${bits}(_value)
  // ...
  @unknown default:
    self._roundSlowPath(rule)
  }
}
```

Making `round(_:)` inlinable but still have a default case is an attempt to get the best of both worlds: if the rounding rule is known at compile time, the call will compile down to a single instruction in optimized builds; and if it dynamically turns out to be a new kind of rounding rule added in Swift 25 (e.g. `.towardFortyTwo`), there's a fallback function, `_roundSlowPath(_:)`, that can handle it.

So what does `_roundSlowPath(_:)` look like? Well, it can't be inlinable, because that would defeat the purpose. It *could* just look like this:

```
@usableFromInline
internal mutating func _roundSlowPath(_ rule: FloatingPointRoundingRule) {
  switch rule {
  case .toNearestOrAwayFromZero:
    _value = Builtin.int_round_FPIEEE${bits}(_value)
  case .toNearestOrEven:
    _value = Builtin.int_rint_FPIEEE${bits}(_value)
```

```
  // ...
  }
}
```

...i.e. exactly the same as `round(_:)` but with no `default` case. That's guaranteed to be up to date if any new cases are added in the future. But it seems a little silly, since it's duplicating code that's in `round(_:)`. We *could* omit cases that have always existed, but there's a better answer:

```
// Slow path for new cases that might have been inlined into an old
// ABI-stable version of round(_:) called from a newer version. If this is
// the case, this non-inlinable function will call into the _newer_ version
// which _will_ support this rounding rule.
@usableFromInline
internal mutating func _roundSlowPath(_ rule: FloatingPointRoundingRule) {
  self.round(rule)
}
```

Because `_roundSlowPath(_:)` isn't inlinable, the version of `round(_:)` that gets called at run time will always be the version implemented in the standard library dylib. And since FloatingPointRoundingRule is *also* defined in the standard library, we know it'll never be out of sync with this version of `round(_:)`.

Maybe some day we'll have special syntax in the language to say "call this method without allowing inlining" to get the same effect, but for now, this Curiously Recursive Inlinable Switch Pattern allows for safe inlining of switches over non-frozen enums with less boilerplate than you might otherwise have. Not none, but less.

(Meta): List of wants and TODOs for this guide

1. Library Organization
    1. What files are where
        1. Brief about CMakeLists
        2. Brief about GroupInfo.json
    2. What tests are where
        1. Furthermore, should there be a split between whitebox tests and blackbox tests?
    3. What benchmarks are where
        1. Furthermore, should there be benchmarks, microbenchmarks, and nanobenchmarks (aka whitebox microbenchmarks)?
    4. What SPIs exist, where, and who uses them
    5. Explain test/Prototypes, and how to use that for rapid (relatively speaking) prototyping
2. Library Concepts
    1. Protocol hierarchy
        1. Customization hooks
    2. Use of classes, COW implementation, buffers, etc
    3. Compatibility, `@available`, etc.
    4. Resilience, ABI stability, `@inlinable`, `@usableFromInline`, etc
    5. Strings and ICU
    6. Lifetimes
        1. withExtendedLifetime, withUnsafe...,
    7. Shims and stubs

3. Coding Standards
    1. High level concerns
    2. Best practices
    3. Formatting
4. Internals
    1. `@inline(__always)` and `@inline(never)`
    2. `@semantics(...)`
    3. Builtins
        1. Builtin.addressof, _isUnique, etc
5. Dirty hacks
    1. Why all the underscores and extra protocols?
    2. How does the `...` ranges work?
6. Frequently Encountered Issues