orphan:

Scope and introduction

This document defines the policy for applying access control modifiers and related naming conventions for the Swift standard library and overlays.

In this document, "stdlib" refers to the core standard library and overlays for system frameworks written in Swift.

Swift has three levels of access control --- private, internal and public. As currently implemented, access control is only concerned with API-level issues, not ABI. The stdlib does not have a stable ABI, and is compiled in "non-resilient" mode with inlining into user code; thus, all stdlib symbols are considered ABI and stdlib clients should be recompiled after *any* change to the stdlib.

public

User-visible APIs should be marked public.

Unfortunately, the compiler has bugs and limitations that the stdlib must work around by defining additional public symbols not intended for direct consumption by users. For example:

```
// Workaround.
public protocol _Collection { ... }

// Symbol intended for use outside stdlib.
public protocol Collection : _Collection { ... }
```

These symbols are hidden using the leading underscore rule.

Because Swift does not yet support a notion of SPI, any implementation details that are shared across the stdlib's various sub-modules must also be public. These names, too, use the leading underscore rule.

To document the reason for marking symbols public, we use comments:

- symbols used in tests:

  ```
  public // @testable
  func _foo() { ... }
  ```

- symbols that are SPIs for the module X:

  ```
  public // SPI(X)
  func _foo() { ... }
  ```

internal

In Swift, *internal* is an implied default everywhere--except within *public* extensions and protocols. Therefore, *internal* should be used explicitly everywhere in the stdlib to avoid confusion.

> **Note**
>
> No declaration should omit an access

To create a "single point of truth" about whether a name is intended for user consumption, the following names should all use the leading underscore rule:

- module-scope *private* and *internal* symbols:

  ```
  var _internalStdlibConstant: Int { ... }
  ```

- *private* and *internal* symbols nested within *public* types:

  ```
  public struct Dictionary {
    var _representation: _DictionaryRepresentation
  }
  ```

private

The *private* modifier cannot be used in the stdlib at least until rdar://17631278 is fixed.

Leading Underscore Rule

Variables, functions and typealiases should have names that start with an underscore:

```
var _value: Int
func _bridgeSomethingToAnything(_ something: AnyObject) -> AnyObject
typealias _InternalTypealias = HeapBuffer<Int, Int>
```

To apply the rule to an initializer, one of its label arguments *or* internal parameter names must start with an underscore:

```
public struct Foo {
  init(_count: Int) {}
  init(_ _otherInitializer: Int) {}
}
```

> **Note**
>
> the identifier that consists of a single underscore _ is not considered to be a name that starts with an underscore. For example, this initializer is public:
>
> ```
> public struct Foo {
>   init(_ count: Int) {}
> }
> ```

The compiler and IDE tools may use the leading underscore rule, combined with additional heuristics, to hide stdlib symbols that users don't need to see.

Users are prohibited to use leading underscores symbols in their own source code, even if these symbols are visible through compiler diagnostics or IDE tools.