Underscored Attributes Reference

WARNING: This information is provided primarily for compiler and standard library developers. Usage of these attributes outside of the Swift monorepo is STRONGLY DISCOURAGED.

The Swift reference has a chapter discussing stable attributes. This document is intended to serve as a counterpart describing underscored attributes, whose semantics are subject to change and most likely need to go through the Swift evolution process before being stabilized.

The attributes are organized in alphabetical order.

`@_alignment(numericValue)`

Allows controlling the alignment of a type.

The alignment value specified must be a power of two, and cannot be less than the "natural" alignment of the type that would otherwise be used by the Swift ABI. This attribute is intended for the SIMD types in the standard library which use it to increase the alignment of their internal storage to at least 16 bytes.

`@_alwaysEmitIntoClient`

Forces the body of a function to be emitted into client code.

Note that this is distinct from `@inline(__always)`; it doesn't force inlining at call-sites, it only means that the implementation is compiled into the module which uses the code.

This means that `@_alwaysEmitIntoClient` definitions are *not* part of the defining module's ABI, so changing the implementation at a later stage does not break ABI.

Most notably, default argument expressions are implicitly `@_alwaysEmitIntoClient`, which means that adding a default argument to a function which did not have one previously does not break ABI.

`@_backDeploy(before: ...)`

Causes the body of a function to be emitted into the module interface to be available for emission into clients with deployment targets lower than the ABI availability of the function. When the client's deployment target is before the function's ABI availability, the compiler replaces calls to that function with a call to a thunk that checks at runtime whether the original library function is available. If the the original is available then it is called. Otherwise, the fallback copy of the function that was emitted into the client is called instead.

For more details, see the pitch thread in the forums.

`@_assemblyVision`

Forces emission of assembly vision remarks for a function or method, showing where various runtime calls and performance impacting hazards are in the code at source level after optimization.

Adding this attribute to a type leads to remarks being emitted for all methods.

`@_borrowed`

Indicates that the conservative access pattern for some storage (a subscript or a property) should use the `_read` accessor instead of `get`.

For more details, see the forum post on Value ownership when reading from a storage declaration.

`@_cdecl("cName")`

Similar to `@_silgen_name` but uses the C calling convention.

This attribute doesn't have very well-defined semantics. Type bridging is not done, so the parameter and return types should correspond directly to types accessible in C. In most cases, it is preferable to define a static method on an `@objc` class instead of using `@_cdecl`.

For potential ideas on stabilization, see Formalizing `@cdecl`.

`@_disfavoredOverload`

Marks an overload that the type checker should try to avoid using. When the expression type checker is considering overloads, it will prefer a solution with fewer `@_disfavoredOverload` declarations over one with more of them.

Use `@_disfavoredOverload` to work around known bugs in the overload resolution rules that cannot be immediately fixed without a source break. Don't use it to adjust overload resolution rules that are otherwise sensible but happen to produce undesirable results for your particular API; it will likely be removed or made into a no-op eventually, and then you will be stuck with an overload set that cannot be made to function in the way you intend.

`@_disfavoredOverload` was first introduced to work around a bug in overload resolution with `ExpressibleByXYZLiteral` types. The type checker strongly prefers to give literals their default type (e.g. `Int` for `ExpressibleByIntegerLiteral`, `String` for `ExpressibleByStringLiteral`, etc.). If an API should prefer some other type, but accept the default too, marking the declaration taking the default type with `@_disfavoredOverload` gives the desired behavior:

```
extension LocalizedStringKey: ExpressibleByStringLiteral { ... }

extension Text {
  // We want `Text("foo")` to use this initializer:
  init(_ key: LocalizedStringKey) { ... }

  // But without @_disfavoredOverload, it would use this one instead,
  // because that lets it give the literal its default type:
  @_disfavoredOverload init<S: StringProtocol>(_ str: S) { ... }
}
```

`@_dynamicReplacement(for: targetFunc(label:))`

Marks a function as the dynamic replacement for another `dynamic` function. This is similar to method swizzling in other languages such as Objective-C, except that the replacement happens at program start (or loading a shared library), instead of at an arbitrary point in time.

For more details, see the forum post on dynamic method replacement.

`@_effects(effectname)`

Tells the compiler that the implementation of the defined function is limited to certain side effects. The attribute argument specifies the kind of side effect limitations that apply to the function including any other functions it calls. This is used to provide information to the optimizer that it can't already infer from static analysis.

Changing the implementation in a way that violates the optimizer's assumptions about the effects results in undefined behavior.

`@_effects(readnone)`

Defines that the function does not have any observable memory reads or writes or any other observable side effects.

This does not mean that the function cannot read or write memory at all. For example, it's allowed to allocate and write to local objects inside the function. For example, the following `readnone` function allocates an array and writes to the array buffer

```
@_effects(readnone)
func lookup(_ i: Int) -> Int {
  let a = [7, 3 ,6, 9]
  return a[i]
}
```

A function can be marked as readnone if two calls of the same function with the same parameters can be simplified to one call (e.g. by the CSE optimization) without changing the semantics of the program. For example,

```
  let a = lookup(i)
  // some other code, including memory writes
  let b = lookup(i)
```

is equivalent to

```
  let a = lookup(i)
  // some other code, including memory writes
  let b = a
```

Some conclusions:

- A `readnone` function must not return a newly allocated class instance.
- A `readnone` function can return a newly allocated copy-on-write object, like an Array, because COW data types conceptually behave like value types.
- A `readnone` function must not release any parameter or any object indirectly referenced from a parameter.
- Any kind of observable side-effects are not allowed, like `print`, file IO, etc.

`@_effects(readonly)`

Defines that the function does not have any observable memory writes or any other observable side effects, beside reading of memory.

Similar to `readnone`, a `readonly` function is allowed to write to local objects.

A function can be marked as `readonly` if it's save to eliminate a call to such a function in case its return value is not used. Example:

```
@_effects(readonly)
func lookup2(_ instance: SomeClass) -> Int {
  let a = [7, 3 ,6, 9]
  return a[instance.i]
}
```

It is legal to eliminate an unused call to this function:

```
_ = lookup2(i)  // can be completely eliminated
```

Note that it would not be legal to CSE two calls to this function, because between those calls the member `i` of the class instance could be modified:

```
  let a = lookup2(instance)
  instance.i += 1
  let b = lookup2(instance)   // cannot be CSE'd with the first call
```

The same conclusions as for `readnone` also apply to `readonly`.

`@_effects(releasenone)`

Defines that the function does not release any class instance.

This effect must be used with care. There are several code patterns which release objects in a non-obvious way. For example:

- A parameter which is passed to an "owned" argument (and not stored), like initializer arguments.
- Assignments, because they release the old value
- COW data types, e.g. Strings. Conceptually they are value types, but internally the keep a reference counted buffer.
- Class references deep inside a hierarchy of value types.

`@_effects(readwrite)`

This effect is not used by the compiler.

`@_effects(notEscaping <selection>)`

Tells the compiler that a function argument does not escape. The *selection* specifies which argument or which "projection" of an argument does not escape. The *selection* consists of the argument name or `self` and an optional projection path.

The projection path consists of field names or one of the following wildcards:

- `class*` : selects any class field, including tail allocated elements
- `value**` : selects any number and any kind of struct, tuple or enum fields/payload-cases.
- `**` : selects any number of any fields

For example:

```
struct Inner {
  let i: Class
}
struct Str {
    let a: Inner
    let b: Class
}

@_effects(notEscaping s.b)    // s.b does not escape, but s.a.i can escape
func foo1(_ s: Str) { ... }

@_effects(notEscaping s.v**)  // s.b and s.a.i do not escape
func foo2(_ s: Str) { ... }

@_effects(notEscaping s.**)   // s.b, s.a.i and all transitively reachable
                              // references from there do not escape
func foo3(_ s: Str) { ... }
```

`@_effects(escaping <from-selection> => <to-selection>)`

Defines that an argument escapes to another argument or to the return value, but not otherwise. The *to-selection* can also refer to `return`.

For example:

```
@_effects(escapes s.b => return)
func foo1(_ s: Str) -> Class {
  return s.b
}

@_effects(escapes s.b => o.a.i)
func foo2(_ s: Str, o: inout Str) {
  o.a.i = s.b
}
```

`@_effects(escaping <from-selection> -> <to-selection>)`

This variant of an escaping effect defines a "non-exclusive" escape. This means that not only the *from-selection*, but also other values can escape to the *to-selection*.

For example:

```
var g: Class

@_effects(escapes s.b -> return)
func foo1(_ s: Str, _ cond: Bool) -> Class {
  return cond ? s.b : g
}
```

`@_exported`

Re-exports all declarations from an imported module.

This attribute is most commonly used by overlays.

```
// module M
public func f() {}

// module N
@_exported import M

// module P
import N
func g() {
  N.f() // OK
}
```

`@_fixed_layout`

Same as `@frozen` but also works for classes.

With `@_fixed_layout` classes, vtable layout still happens dynamically, so non-public virtual methods can be removed, new virtual methods can be added, and existing virtual methods can be reordered.

`@_hasInitialValue`

Marks that a property has an initializing expression.

This information is lost in the swiftinterface, but it is required as it results in a symbol for the initializer (if a class/struct `init` is inlined, it will call initializers for properties that it doesn't initialize itself). This information is necessary for correct TBD file generation.

`@_hasMissingDesignatedInitializers`

Indicates that there may be designated initializers that are not printed in the swiftinterface file for a particular class.

This attribute is needed for the initializer model to maintain correctness when library evolution is enabled. This is because a class may have non-public designated initializers, and Swift allows the inheritance of convenience initializers if and only if the subclass overrides (or has synthesized overrides) of every designated initializer in its superclass. Consider the following code:

```
// Lib.swift
open class A {
  init(invisible: ()) {}

  public init(visible: ()) {}
  public convenience init(hi: ()) { self.init(invisible: ()) }
}

// Client.swift
class B : A {
  var x: String

  public override init(visible: ()) {
    self.x = "Garbage"
    super.init(visible: ())
  }
}
```

In this case, if `B` were allowed to inherit the convenience initializer `A.init(invisible:)` then an instance created via `B(hi: ())` would fail to initialize `B.x` resulting in a memory safety hole. What's worse is there is no way to close this safety hole because the user cannot override the invisible designated initializer because they lack sufficient visibility.

`@_hasStorage`

Marks a property as being a stored property in a swiftinterface.

For `@frozen` types, the compiler needs to be able to tell whether a particular property is stored or computed to correctly perform type layout.

```
@frozen struct S {
  @_hasStorage var x: Int { get set } // stored
  var y: Int { get set } // computed
}
```

`@_implementationOnly`

Used to mark an imported module as an implementation detail. This prevents types from that module being exposed in API (types of public functions, constraints in public extension etc.) and ABI (usage in `@inlinable` code).

`@_implements(ProtocolName, Requirement)`

An attribute that indicates that a function with one name satisfies a protocol requirement with a different name. This is especially useful when two protocols declare a requirement with the same name, but the conforming type wishes to offer two separate implementations.

```
protocol P { func foo() }

protocol Q { func foo() }
```

```
struct S : P, Q {
  @_implements(P, foo())
  func foo_p() {}
  @_implements(Q, foo())
  func foo_q() {}
}
```

`@_implicitSelfCapture`

Allows access to `self` inside a closure without explicitly capturing it, even when `Self` is a reference type.

```
class C {
  func f() {}
  func g(_: @escaping () -> Void) {
    g({ f() }) // error: call to method 'f' in closure requires explicit use of
'self'
  }
  func h(@_implicitSelfCapture _: @escaping () -> Void) {
    h({ f() }) // ok
  }
}
```

`@_inheritActorContext`

(Note that it is "inherit", not "inherits", unlike below.)

Marks that a `@Sendable async` closure argument should inherit the actor context (i.e. what actor it should be run on) based on the declaration site of the closure. This is different from the typical behavior, where the closure may be runnable anywhere unless its type specifically declares that it will run on a specific actor.

`@_inheritsConvenienceInitializers`

An attribute that signals that a class declaration inherits its convenience initializers from its superclass. This implies that all designated initializers -- even those that may not be visible in a swiftinterface file -- are overridden. This attribute is often printed alongside `@_hasMissingDesignatedInitializers` in this case.

`@_marker`

Indicates that a protocol is a marker protocol. Marker protocols represent some meaningful property at compile-time but have no runtime representation.

For more details, see SE-0302, which introduces marker protocols. At the moment, the language only has one marker protocol: `Sendable`.

Fun fact: Rust has a very similar concept called marker traits, including one called `Send`, which inspired the design of `Sendable`.

`@_nonEphemeral`

Marks a function parameter that cannot accept a temporary pointer produced from an inout-to-pointer, array-to-pointer, or string-to-pointer conversion. Such a parameter may only accept a pointer that is guaranteed to outlive the duration of the function call.

Attempting to pass a temporary pointer to an `@_nonEphemeral` parameter will produce a warning. This attribute is primarily used within the standard library on the various `UnsafePointer` initializers to warn users about the undefined behavior caused by using a temporary pointer conversion as an argument:

```
func baz() {
  var x = 0

  // warning: Initialization of 'UnsafePointer<Int>' results in a dangling pointer
  let ptr = UnsafePointer(&x)

  // warning: Initialization of 'UnsafePointer<Int>' results in a dangling pointer
  let ptr2 = UnsafePointer([1, 2, 3])
}
```

The temporary pointer conversion produces a pointer that is only guaranteed to be valid for the duration of the call to the initializer, and becomes invalid once the call ends. So the newly created `UnsafePointer` will be dangling.

One exception to this is that inout-to-pointer conversions on static stored properties and global stored properties produce non-ephemeral pointers, as long as they have no observers:

```
var global = 0

struct S {
  static var staticVar = 0
}

func baz() {
  let ptr = UnsafePointer(&global) // okay
  let ptr2 = UnsafePointer(&S.staticVar) // okay
}
```

Additionally, if they are of a tuple or struct type, their stored members without observers may also be passed inout as non-ephemeral pointers.

For more details, see the educational note on temporary pointer usage.

`@_nonoverride`

Marks a declaration that is not an override of another. When the `-warn-implicit-overrides` flag is used, a warning is issued when a protocol restates a requirement from another protocol it refines without annotating the declaration with either `override` or `@_nonoverride`.

An `override` annotation causes the overriding declaration to be treated identically to the overridden declaration; a conforming type can only provide one implementation ("witness"). Restating a protocol requirement and then marking it as an `override` is generally only needed to help associated type inference, and many `override` annotations correlate closely with ABI FIXMEs.

Meanwhile, `@_nonoverride` is the "opposite" of `override`, allowing two protocol requirements to be treated independently; a conforming type can provide a distinct witness for each requirement (for example, by using `@_implements`). Use `@_nonoverride` when semantics differ between the two requirements. For example, `BidirectionalCollection.index(_:offsetBy:)` allows negative offsets, while `Collection.index(_:offsetBy:)` does not, and therefore the former is marked `@_nonoverride`.

The `@_nonoverride` annotation can also be specified on class members in addition to protocol members. Since it is the "opposite" of `override`, it can be used to suppress "near-miss" diagnostics for declarations that are similar to but not meant to override another declaration, and it can be used to intentionally break the override chain, creating an overload instead of an override.

This attribute and the corresponding `-warn-implicit-overrides` flag are used when compiling the standard library and overlays.

`@_nonSendable`

There is no clang attribute to add a Swift conformance to an imported type, but there *is* a clang attribute to add a Swift attribute to an imported type. So `@Sendable` (which is not normally allowed on types) is used from clang headers to indicate that an unconstrained, fully available `Sendable` conformance should be added to a given type, while `@_nonSendable` indicates that an unavailable `Sendable` conformance should be added to it.

`@_nonSendable` can have no options after it, in which case it "beats" `@Sendable` if both are applied to the same declaration, or it can have `(_assumed)` after it, in which case `@Sendable` "beats" it. `@_nonSendable(_assumed)` is intended to be used when mass-marking whole regions of a header as non-`Sendable` so that you can make spot exceptions with `@Sendable`.

`@_objc_non_lazy_realization`

Marks a class as being non-lazily (i.e. eagerly) realized.

This is used for declarations which may be statically referenced and wouldn't go through the normal lazy realization paths. For example, the empty array class must be non-lazily realized, because empty arrays are statically allocated. Otherwise, passing the empty array object to other code without triggering realization could allow for the unrealized empty array class to be passed to ObjC runtime APIs which only operate on realized classes, resulting in a crash.

`@_optimize([none|size|speed])`

Controls the compiler's optimization mode. This attribute is analogous to the command-line flags `-Onone`, `-Osize` and `-Ospeed` respectively, but limited to a single function body.

`@_optimize(none)` is handy for diagnosing and reducing compiler bugs as well as improving debugging in Release builds.

`@_originallyDefinedIn(module: "ModuleName", availabilitySpec...)`

Marks a declaration as being originally defined in a different module, changing the name mangling. This can be used to move declarations from a module to one of the modules it imports without breaking clients.

Consider the following example where a framework ToasterKit needs to move some APIs to a lower-level framework ToasterKitCore. Here are the necessary changes:

1. Add a linker flag `-reexport_framework ToasterKitCore` for ToasterKit. This ensures all symbols defined in ToasterKitCore will be accessible during runtime via ToasterKit, so existing apps continue to run.
2. In ToasterKit, use `@_exported import ToasterKitCore`. This ensures existing source code that only imports ToasterKit continues to type-check.
3. Move the necessary declarations from ToasterKit to ToasterKitCore. The moved declaration should have two attributes:
   - `@available` indicating when the declaration was introduced in ToasterKit.
   - `@_originallyDefinedIn` indicating the original module and when the declaration was moved to ToasterKitCore.

   ```
   @available(toasterOS 42, *)
   @_originallyDefinedIn(module: "ToasterKit", toasterOS 57)
   enum Toast {
   case underdone
   case perfect
   case burnt
   }
   ```

4. Add Swift compiler flags `-Xfrontend -emit-ldadd-cfile-path -Xfrontend /tmp/t.c` to ToasterKitCore's build settings. Add the emitted `/tmp/t.c` file to ToasterKit's compilation. This ensures when an app is built for deployment targets prior to the symbols' move, the app will look for these symbols in ToasterKit instead of ToasterKitCore.

More generally, mutliple availabilities can be specified, like so:

```
@available(toasterOS 42, bowlOS 54, mugOS 54, *)
@_originallyDefinedIn(module: "ToasterKit", toasterOS 57, bowlOS 69, mugOS 69)
enum Toast { ... }
```

`@_private(sourceFile: "FileName.swift")`

Fully bypasses access control, allowing access to private declarations in the imported module. The imported module needs to be compiled with `-Xfrontend -enable-private-imports` for this to work.

`@_semantics("uniquely.recognized.id")`

Allows the optimizer to make use of some key invariants in performance critical data types, especially `Array`. Since the implementation of these data types is written in Swift using unsafe APIs, without these attributes the optimizer would need to make conservative assumptions.

Changing the implementation in a way that violates the optimizer's assumptions about the semantics results in undefined behavior.

`@_show_in_interface`

Shows underscored protocols from the standard library in the generated interface.

By default, SourceKit hides underscored protocols from the generated swiftinterface (for all modules, not just the standard library), but this attribute can be used to override that behavior for the standard library.

`@_silgen_name("cName")`

Changes the symbol name for a function, similar to an ASM label in C, except that the platform symbol mangling (leading underscore on Darwin) is maintained.

Since this has label-like behavior, it may not correspond to any declaration; if so, it is assumed that the function is implemented in C.

A function defined by `@_silgen_name` is assumed to use the Swift ABI.

For more details, see the Standard Library Programmer's Manual.

`@_specialize(...)`

Forces generation of a specialized implementation for a generic declaration.

See Generics.rst for more details.

`@_specializeExtension`

Allows extending `@usableFromInline` internal types from foreign modules. Consider the following example involving two modules:

```
// Module A
@usableFromInline
internal struct S<T> { /* ... */ }

// Module B
import A

@_specializeExtension
extension S { // OK
  // add methods here
}

extension S /* or A.S */ { // error: cannot find 'S' in scope
}
```

This ability can be used to add specializations of existing methods in downstream libraries when used in conjunction with `@_specialize`.

```
// Module A
@usableFromInline
internal struct S<T> {
  @inlinable
  internal func doIt() { /* body */ }
}
```

```
// Module B
import A

@_specializeExtension
extension S { // ok
  @_specialize(exported: true, target: doIt(), where T == Int)
  public func specializedDoIt() {}
}

// Module C
import A
import B

func f(_ s: S<Int>) {
  s.doIt() // will call specialized version of doIt() where T == Int from B
}
```

`@_spi(spiName)`

Marks a declaration as SPI (System Programming Interface), instead of API. Modules exposing SPI and using library evolution generate an additional `.private.swiftinterface` file (with `-emit-private-module-interface-path`) in addition to the usual `.swiftinterface` file. This private interface exposes both API and SPI.

Clients can access SPI by marking the import as `@_spi(spiName) import Module`. This design makes it easy to find out which clients are using certain SPIs by doing a textual search.

`@_staticInitializeObjCMetadata`

Indicates that a static initializer should be emitted to register the Objective-C metadata when the image is loaded, rather than on first use of the Objective-C metadata.

This attribute is inferred for `NSCoding` classes that won't have static Objective-C metadata or have an `@NSKeyedArchiveLegacy` attribute.

`@_transparent`

Marks a function to be "macro-like", i.e., it is guaranteed to be inlined in debug builds.

See TransparentAttr.md for more details.

`@_typeEraser(Proto)`

Marks a concrete nominal type as one that implements type erasure for a protocol `Proto`.

A type eraser has the following restrictions:

1. It must be a concrete nominal type.
2. It must not have more restrictive access than `Proto`.
3. It must conform to `Proto`.
4. It must have an initializer of the form `init<T: Proto>(erasing: T)`.

- Other generic requirements are permitted as long as the `init` can always be called with a value of any type conforming to `Proto`.
- The `init` cannot have more restrictive access than `Proto`.

This feature was designed to be used for compiler-driven type erasure for dynamic replacement of functions with an opaque return type.

`@_weakLinked`

Allows a declaration to be weakly-referenced, i.e., any references emitted by client modules to the declaration's symbol will have weak linkage. This means that client code will compile without the guarantee that the symbol will be available at runtime. This requires a dynamic safety check (such as using `dlsym (3)`); otherwise, accessing the symbol when it is unavailable leads to a runtime crash.

This is an unsafe alternative to using `@available`, which is statically checked. If the availability of a library symbol is newer than the deployment target of the client, the symbol will be weakly linked, but checking for `@available` and `#(un)available` ensures that a symbol is not accessed when it is unavailable.

`@_unsafeMainActor`, `@_unsafeSendable`

Marks a parameter's (function) type as `@MainActor` (`@Sendable`) in Swift 6 and within Swift 5 code that has adopted concurrency, but non-`@MainActor` (non-`@Sendable`) everywhere else.

See the forum post on Concurrency in Swift 5 and 6 for more details.

`@_noImplicitCopy`

Marks a var decl as a variable that must be copied explicitly using the builtin function Builtin.copy.

`@_noAllocation`, `@_noLocks`

These attributes are performance annotations. If a function is annotated with such an attribute, the compiler issues a diagnostic message if the function calls a runtime function which allocates memory or locks, respectively. The `@_noLocks` attribute implies `@_noAllocation` because a memory allocation also locks.

`@_unavailableFromAsync`

Marks a synchronous API as being unavailable from asynchronous contexts. Direct usage of annotated API from asynchronous contexts will result in a warning from the compiler.

`@_unsafeInheritExecutor`

This `async` function uses the pre-SE-0338 semantics of unsafely inheriting the caller's executor. This is an underscored feature because the right way of inheriting an executor is to pass in the required executor and switch to it. Unfortunately, there are functions in the standard library which need to inherit their caller's executor but cannot change their ABI because they were not defined as `@_alwaysEmitIntoClient` in the initial release.

`@_spi_available(platform, version)`

Like `@available`, this attribute indicates a decl is available only as an SPI. This implies several behavioral changes comparing to regular `@available`:

1. Type checker diagnoses when a client accidently exposes such a symbol in library APIs.
2. When emitting public interfaces, `@_spi_available` is printed as `@available(platform, unavailable)`.
3. ClangImporter imports ObjC macros `SPI_AVAILABLE` and `__SPI_AVAILABLE` to this attribute.

`_local`

A distributed actor can be marked as "known to be local" which allows avoiding the distributed actor isolation checks. This is used for things like `whenLocal` where the actor passed to the closure is known-to-be-local, and similarly a `self` of obtained from an *isolated* function inside a distributed actor is also guaranteed to be local by construction.