Name Translation from C to Swift

This document gives a high-level description of how C and Objective-C declarations are translated to Swift, with particular focus on how names are adjusted. It is not attempting to be a *complete* description of everything the compiler does except with regards to how *names* are transformed; even there, some special cases that only apply to Apple's SDKs have been omitted. The example code shown is for illustrative purposes and does not always include all parts of an imported API's interface.

Word boundaries

Several forms of name translation are defined in terms of word boundaries. The word-splitting algorithm used by the Swift compiler is as follows: there is a boundary after

1. An underscore ("_").
2. A series of two or more uppercase ASCII characters and the suffix "s", "es", or "ies" (e.g. "URLs", "VAXes")...unless the last uppercase letter is "I" and the suffix is "s", in which case it's just as likely to be an acronym followed by "Is" (i.e. "URLIs" is treated as "URL Is").
3. A series of two or more uppercase ASCII characters followed by an uppercase ASCII character and then a lowercase ASCII character ("XMLReader" becomes "XML Reader").
4. A series of two or more uppercase ASCII characters followed by a non-ASCII-alphabetic character. ("UTF8" becomes "UTF 8")
5. A series of two or more uppercase ASCII characters at the end of the string.
6. An uppercase ASCII character and any number of non-ASCII-uppercase, non-underscore characters ("ContrivedExample" becomes "Contrived Example").
7. Any number of non-ASCII-uppercase, non-underscore characters ("lowercase_example" becomes "lowercase _ example").

Enums

1. Anonymous? Import elements as constants of the underlying type, *except* that Int is used for an inferred underlying type if all the cases fit in an Int32, since integer conversions are explicit in Swift.
2. `ns_error_domain` attribute? Import as an error struct (see below)
3. `flag_enum` attribute? Import as an option set struct (see below)
4. `enum_extensibility` attribute? Import as an `@objc` enum, whether open or closed
5. Otherwise, import as a RawRepresentable struct *without renaming constants* (this is probably an oversight but would be source-breaking to change)

An enum declared as `typedef enum { … } MyEnum;` is not considered "anonymous"; the typedef name is used instead. This interpretation has precedent in the C++ linkage rules, which have a similar special case for typedefs of unnamed enums (C++17 [dcl.typedef]p9).

`@objc` enums

```
enum TimeOfDay __attribute__((enum_extensibility(open))) : long {
  TimeOfDayMorning,
  TimeOfDayAfternoon,
  TimeOfDayNight,
  TimeOfDayEvening = TimeOfDayNight
};
// Most commonly seen as NS_ENUM and NS_CLOSED_ENUM.
```

```
@objc enum TimeOfDay: Int {
  init?(rawValue: Int)
  var rawValue: Int { get }

  case morning
  case afternoon
  case night

  static var evening: TimeOfDay { get }
}
```

- Enum name is enum name.
- Underlying type becomes raw type.
- Case names follow the rules for "enum-style prefix stripping" below.
- `init?(rawValue:)` currently does not check cases, but probably should for "closed" enums (Clang's equivalent of `@frozen`)

In order to deal with cases with the same underlying value, the importer picks a set of *canonical cases* by walking the enum and recording the first *available* enumerator with a particular value. (Note that deprecated cases are still considered available unless they were deprecated for a platform older than Swift's earliest deployment targets, OS X 10.9 and iOS 7.) Any "non-canonical" cases are imported as computed properties instead.

Error enum structs

```
enum VagueFailureCode __attribute__((ns_error_domain(VagueFailureDomain))) : long {
  VagueFailureBadness,
  VagueFailureWorseness,
  VagueFailureWorstness
};
// Most commonly seen as NS_ERROR_ENUM.
```

```
struct VagueFailure: Error {
  @objc enum Code: Int {
    init?(rawValue: Int)
    var rawValue: Int { get }

    case badness
    case worseness
    case worstness

    typealias ErrorType = VagueFailure
  }

  static var badness: VagueFailure.Code { get }
  static var worseness: VagueFailure.Code { get }
  static var worstness: VagueFailure.Code { get }

  static var errorDomain: String { get }
}
```

- Struct name is enum name, dropping the suffix "Code" if present.
- Original enum type is mapped to the nested "Code" enum using the rules for "`@objc` enums".
- Struct gets an `errorDomain` member populated from the `ns_error_domain` attribute.
- Struct gets aliases for the enum cases as static properties for easier use in `catch` clauses.

Option set structs

```
enum PetsAllowed __attribute__((flag_enum)) : long {
  PetsAllowedNone = 0,
  PetsAllowedDogs = 1 << 0,
  PetsAllowedCats = 1 << 1
};
// Most commonly seen as NS_OPTIONS.
```

```
struct PetsAllowed: OptionSet {
  init(rawValue: Int)
  var rawValue: Int

  static var dogs: PetsAllowed { get }
  static var cats: PetsAllowed { get }
}
```

- Struct name is enum name.
- Underlying type becomes raw type.
- Cases are imported as static properties.
- Case names follow the rules for "enum-style prefix stripping" below.
- Cases with a raw value of `0` are **not imported** unless the case has an explicit `swift_name` attribute. (The intent is to use `[]` instead in Swift, representing "no options".)

Enum-style prefix stripping

In C, enumerators (enum cases) aren't namespaced under their enum type, so their names have to make sense in a global context. The Swift compiler attempts to recognize several common idioms for ASCII-based names in order to translate these into idiomatic Swift members.

1. Collect all *available, non-deprecated* enum cases *without custom names*. If there are no such cases, collect all cases without custom names, whether available or not.
2. Find the common word-boundary prefix **CP** of these cases.
3. If **CP** starts with "k" followed by an uppercase letter, or if it's *just* "k" and none of the cases have a non-identifier-start character immediately after the 'k', treat that as meaning "constant" and ignore it for the next step.
4. Find the common word-boundary prefix **EP** of **CP** and the type's original C name (rather than its Swift name).
5. If the next word of **CP** after **EP** is
    - the next word of the type's original C name minus "s" ("URL" vs. "URLs")
    - the next word of the type's original C name minus "es" ("Address" vs. "Addresses")
    - the next word of the type's original C name with "ies" replaced by "y" ("Property" vs. "Properties")

add the next word of **CP** to **EP**.

6. If the next word of **CP** after **EP** is an underscore ("MyEnum_FirstCase" vs. "MyEnum"), add the underscore to **EP**.
7. If "k" was dropped from **CP** in step 3, add it back to the front of **EP**.
8. For any case without a custom name, if it starts with **EP**, drop **EP** from that case's name when importing it. (Deprecated or unavailable cases may not start with **EP** depending on step 1.)
9. ASCII-lowercase the first word of the remaining name if it starts with an uppercase ASCII character.

   There's a bug in this step where the special case for "Is" is missing, so "URLIs" will be lowercased to "urlis".

`swift_wrapper` typedefs

The `swift_wrapper` Clang attribute allows importing a typedef as a RawRepresentable struct type; it also causes global constants that use that type name to be imported as members of the new struct type.

```
typedef NSString * _Nonnull SecretResourceID __attribute__((swift_wrapper(struct)));

extern SecretResourceID const SecretResourceTreasureChest;
extern SecretResourceID const SecretResourceBankVault;

// Usually seen as NS_TYPED_ENUM and NS_TYPED_EXTENSIBLE_ENUM.
```

```
struct SecretResourceID: RawRepresentable, Hashable {
  typealias RawValue = String

  init(_ rawValue: String)
  init(rawValue: String)
  var rawValue: String { get }
}

extension SecretResourceID {
  static var treasureChest: SecretResource { get }
  static var bankVault: SecretResource { get }
}
```

- If the underlying type conforms to Equatable or Hashable, or is bridged to Objective-C, the wrapper type will too.
- The unlabeled `init(_:)` is only created when the Clang attribute is `swift_wrapper(struct)` rather than `swift_wrapper(enum)` .
- The constant names are imported using a simplified version of enum-style prefix stripping unless they have a custom name:

  1. If the constant name starts with "k" followed by an uppercase letter, treat that as meaning "constant" and ignore it for the next step.
  2. Find the common word-boundary prefix **P** of the constant name and the typedef's original C name (rather than its Swift name).

3. If "k" was dropped from the constant name in step 1, add it back to the front of **P**.
4. Drop **P** from the constant's name.
5. ASCII-lowercase the first word of the remaining name if it starts with an uppercase ASCII character.

This default translation may result in a name that is not a valid identifier, in which case a custom name must be applied in order to reference the constant from Swift.

NSNotificationName

On Apple platforms, whenever Foundation is imported, constants with the type "NSNotificationName" additionally have the suffix "Notification" stripped before performing the above rules unless they have a custom name. Global NSString constants whose name ends in "Notification" will also automatically be treated as if they were declared with the type NSNotificationName unless they have a custom name.

Objective-C Protocols

Protocols in Objective-C are normally in a separate namespace from the "ordinary" identifier namespace used by typedefs and classes. Swift does not have separate namespaces, so if the protocol has the same name as another declaration in the same module, the suffix "Protocol" is appended. (Example: NSObjectProtocol in the ObjectiveC module.)

CF Types

"Core Foundation" is a C-based object-oriented system with strong conventions built around pointers to opaque structs. Creating new Core Foundation types is not generally supported, but Swift will recognize those in Apple's SDKs. If a struct has the `objc_bridge`, `objc_bridge_mutable`, or `objc_bridge_related` Clang attributes, it will be treated as a CF type and a typedef of a pointer to that struct will be imported as a class in Swift. The suffix "Ref" will be dropped from the class's name if present unless doing so would conflict with another declaration in the same module as the typedef.

If the class name contains the word "Mutable" exactly once per the usual word-boundary rules, a corresponding class name without the word "Mutable" will be used as the superclass if present. Otherwise, the CF type is taken to be a root object.

Additionally, typedefs for `void *` or `const void *` that are themselves annotated with `objc_bridge` will be treated as CFTypeRef-like and imported as `Any` rather than `Unsafe[Mutable]RawPointer`.

If a typedef's underlying type is itself a "CF pointer" typedef, the "alias" typedef will be imported as a regular typealias, with the suffix "Ref" still dropped from its name (if present) unless doing so would conflict with another declaration in the same module as the typedef.

Objective-C Methods

Objective-C methods are classified into one of five categories:

1. Instance methods that are part of the `init` method family whose selector's first word is "init" are considered "normal" initializers.
2. Class methods that return the type of the containing class or `instancetype` are considered "factory initializers" if a "leading type match" succeeds against the selector, using the containing type (see CToSwiftNameTranslation-OmitNeedlessWords.md).

   As an exception, no-argument factory methods cannot have any additional text after the matching portion.

3. Accessor methods are associated with a property declaration (`@property`).
4. Subscript methods have one of Objective-C's special subscript selectors (`objectAtIndexedSubscript:` , `objectForKeyedSubscript:` , `setObject:atIndexedSubscript:` , or `setObject:forKeyedSubscript:`).
5. Other methods are imported as plain methods.

If a method overrides a superclass or matches a method in an adopted protocol, the Swift name of the "overridden" method will be used for consistency. If there's more than one such name, one is chosen arbitrarily.

Normal Initializers

The base name of an initializer is always the special name "init". Any leading "init" is dropped from the original selector. If the next word in the first selector piece is "With":

1. Drop "With".
2. If the remaining text is longer than one letter, the first letter is an ASCII uppercase character, and the second letter is *not* an ASCII uppercase character, downcase the first letter.
3. If the result is a Swift keyword, undo the previous downcasing and insert "with" in front instead.
4. Set this as the first argument name.

Whether or not the initializer will throw is determined according to the "error handling" logic described below. If the initializer is considered throwing, an NSError out-parameter may be removed from the argument list.

At this point, the method name goes through the "omit needless words" process described in CToSwiftNameTranslation-OmitNeedlessWords.md. This process attempts to omit superfluous type names and other words that make a name fit the Cocoa Naming Guidelines for Objective-C but not the Swift API Design Guidelines. For initializers, this is based on

- the argument types, if any, and whether they'll have default values
- the first argument name that was just computed and the remaining selector pieces, if any

If the resulting name has a first argument name but there are no arguments in the original method, a dummy parameter with type `Void` is added.

A normal initializer in a class is considered designated (non-convenience) if it is marked with the `objc_designated_initializer` attribute, *or* if the class has *no* initializers marked with the `objc_designated_initializer` attribute. Otherwise, it is considered convenience.

```
@interface CorporateEmployee : NSObject
- (instancetype)initWithName:(NSString *)name manager:(nullable CorporateEmployee
*)manager __attribute__((objc_designated_initializer));
- (instancetype)initCEOWithName:(NSString *)name;
- (instancetype)initTimCookHimself;
@end

// Usually seen as NS_DESIGNATED_INITIALIZER.
```

```
class CorporateEmployee {
  init(name: String, manager: CorporateEmployee?)
  convenience init(ceoWithName: String)
```

```
  convenience init(timCookHimself: ())
}
```

If an initializer in a class matches a requirement in a protocol adopted by the class, it is imported as `required`.

Factory initializers

The base name of a factory initializer is always the special name "init". The "leading type match" portion of the first selector piece is dropped. If the next word in the first selector piece is "With":

1. Drop "With".
2. If the remaining text is longer than one letter, the first letter is an ASCII uppercase character, and the second letter is *not* an ASCII uppercase character, downcase the first letter.
3. If the result is a Swift keyword, undo the previous downcasing and insert "with" in front instead.
4. Set this as the first argument name.

Whether or not the initializer will throw is determined according to the "error handling" logic described below. If the initializer is considered throwing, an NSError out-parameter may be removed from the argument list.

At this point, the method name goes through the "omit needless words" process described in CToSwiftNameTranslation-OmitNeedlessWords.md. This process attempts to omit superfluous type names and other words that make a name fit the Cocoa Naming Guidelines for Objective-C but not the Swift API Design Guidelines. For factory initializers, this is based on

- the argument types, if any, and whether they'll have default values
- the first argument name that was just computed and the remaining selector pieces, if any

Factory initializers are considered convenience initializers if they have a return type of `instancetype`, and a special "non-inherited factory initializer" otherwise.

```
@interface SpellBook : NSObject
+ (instancetype)spellBookWithAuthor:(NSString *)authorName;
+ (nullable SpellBook *)spellBookByTranslatingAncientText:(AncientText *)text error:
(NSError **)error;
@end
```

```
class SpellBook: NSObject {
  convenience init(author authorName: String)
  /* non-inherited */ init(byTranslating text: AncientText) throws
}
```

If a factory initializer turns out to have the same imported name as an available designated initializer, or a non-inherited factory initializer has the same imported name as an available convenience initializer, the factory initializer is marked unavailable to resolve ambiguities. If a convenience factory initializer has the same imported name as an available convenience initializer, the initializer with more restrictive availability is marked unavailable. In the case of a tie, the convenience factory initializer is the one marked unavailable.

Property accessors

A method that has an associated property declaration is imported with a type that matches the property's type and made into a property accessor. See the section on Objective-C properties below.

Subscript accessors

Methods named `objectAtIndexedSubscript:`, `objectForKeyedSubscript:`, `setObject:atIndexedSubscript:`, or `setObject:forKeyedSubscript:` are considered subscript accessors. If a getter method exists on a type, it will be imported as a subscript; a setter method must be paired with a getter method in the same class/protocol, or in a superclass.

If a subscript getter and setter disagree about the optionality of the element type, the subscript's element type will be imported as implicitly-unwrapped optional. If the getter and setter element types differ in any other way, the subscript will not be imported. If the getter and setter *index* types differ in any way, the subscript will be imported as read-only.

These two fallbacks should have been the same, but aren't.

As a special case, NSDictionary's subscript is imported with a key type of `NSCopying` rather than `Any` so that it has an index type compatible with NSMutableDictionary's subscript setter.

Normal methods

All other Objective-C methods are imported as...methods. Whether or not the method will throw is determined according to the "error handling" logic described below. If the method is considered throwing, an NSError out-parameter may be removed from the argument list.

If the first selector piece of the method is empty, the method is not imported into Swift.

After any error handling transformations, the method name goes through the "omit needless words" process described in CToSwiftNameTranslation-OmitNeedlessWords.md. This process attempts to omit superfluous type names and other words that make a name fit the Cocoa Naming Guidelines for Objective-C but not the Swift API Design Guidelines. For methods, this is based on

- the method base name
- the method return type
- the argument types, if any, and whether they'll have default values
- the first argument name that was just computed and the remaining selector pieces, if any
- the *local* parameter name for the first parameter
- the containing class or protocol
- the set of properties and property-like methods present on the containing class and categories in the same module as the class, including those inherited from superclasses and their categories. (A property-like method is a no-argument method with a non-`void`, non-`instancetype` return type.)

```
@interface UIColor : NSObject
- (UIColor *)colorWithAlphaComponent:(CGFloat)alpha;
- (UIColor *)resolvedColorWithTraitCollection:(UITraitCollection *)traitCollection;
@end
```

```
class UIColor {
  func withAlphaComponent(_ alpha: CGFloat) -> UIColor
  func resolvedColor(with traitCollection: UITraitCollection) -> UIColor
}
```

```
@interface UIView : UIResponder
- (CGPoint)convertPoint:(CGPoint)point toView:(nullable UIView *)view;

@property (readonly) NSArray<NSLayoutConstraint *> *constraints;
- (void)addConstraint:(NSLayoutConstraint *)constraint;
@end
```

```
class UIView {
  func convert(_ point: CGPoint, to view: UIView?) -> CGPoint

  var constraints: [NSLayoutConstraint] { get }
  func addConstraint(_ constraint: NSLayoutConstraint) // rather than add(_:)
}
```

Error handling

Certain methods with NSError out-parameters are imported as throwing methods in Swift. The conditions for this are as follows:

- The method has a parameter with the type `NSError * __autoreleasing *` or `NSError * __unsafe_unretained *`, called the *NSError out-parameter*. Note that `__autoreleasing` is the default for indirect pointers under Objective-C ARC.
- The NSError out-parameter is the last parameter in the method other than parameters with block type.
- The method provides a way to indicate failure:
    - an explicit `swift_error` attribute with a value other than `none`, or
    - a return type of `BOOL` or `Boolean`, or
    - a return type that will be imported as Optional

  Note that the built-in `bool` (`_Bool`) type is *not* considered a boolean type for the purposes of inferring "throws".
- The method does not have a `swift_error` attribute with a value of `none`.

If the method in question is not going to be imported as an initializer, and the NSError out-parameter is the first parameter, and the first selector piece ends with "AndReturnError" or "WithError", the matching suffix is dropped from the base name unless the resulting base name would be a Swift keyword. If the NSError out-parameter is *not* the first parameter, the corresponding selector piece is dropped entirely. The modified selector is then compared against other methods in the class. If there are no other methods in the class matching the new selector, the name change is accepted; otherwise, the original selector is used.

```
- (BOOL)performDelicateActivity:(NSOperation *)operation error:(NSError **)error;
- (BOOL)performDelicateActivityAndReturnError:(NSError **)error activityBody:
(BOOL(^)(void))activityBody;
- (BOOL)performTheUsualActivityWithError:(NSError **)error;
- (BOOL)performYetAnotherActivity:(NSError **)error;
```

```
func performDelicateActivity(_ operation: NSOperation) throws
func performDelicateActivity(_ activityBody: () -> Bool) throws
func performTheUsualActivity() throws
func performYetAnotherActivity() throws
```

If the original selector was used and no "AndReturnError" or "WithError" suffix-stripping was attempted, the imported method will have a dummy parameter with type `Void` in place of the NSError out-parameter. In all other cases (a modified selector or a selector with a suffix that could have been stripped), the NSError out-parameter is removed from the method.

```
- (nullable NSString *)fetchDisplayNameOfResource:(NSURL *)resource;
- (nullable NSString *)fetchDisplayNameOfResource:(NSURL *)resource error:(NSError
**)error;
- (nullable NSString *)fetchDisplayNameOfMyFavoriteSong;
- (nullable NSString *)fetchDisplayNameOfMyFavoriteSongAndReturnError:(NSError
**)error;
```

```
func fetchDisplayName(ofResource: URL) -> String?
func fetchDisplayName(ofResource: URL, error: ()) throws -> String
func fetchDisplayNameOfMyFavoriteSong() -> String?
func fetchDisplayNameOfMyFavoriteSongAndReturnError() throws -> String
```

The return type will be transformed according to the kind of failure indication:

- `swift_error(nonnull_error)` : no transformation
- `swift_error(null_result)` (default for Optional return types): return type becomes non-optional
- `swift_error(zero_result)` (default for `BOOL` and `Boolean`): `BOOL` and `Boolean` become `Void` ; other return types are unchanged
- `swift_error(nonzero_result)` : return type becomes `Void`

Default argument values

Certain method parameters are automatically considered to have default argument values when imported into Swift. This inference uses the following algorithm:

1. If the first word of the method base name is "set", the first parameter of that method never has a default value.
2. A final argument that is a nullable function or block pointer defaults to `nil` .
3. A nullable argument of type `NSZone *` defaults to `nil` .
4. An argument whose type is an option set enum (see above) where the C name of the enum contains the word "options" defaults to `[]` .

   This mistakenly does not check the names of typedefs that wrap anonymous enums, which are supposed to be treated as the name of the enum.
5. An argument whose Objective-C type is an NSDictionary defaults to `nil` if the NSDictionary is nullable and `[:]` if the NSDictionary is non-nullable, under the following conditions:

- the argument label contains the word "options" or "attributes", or the two words "user info" one after another
- the argument label is empty and the method base name ends with the word "options" or "attributes", or the two words "user info" one after another

Objective-C Properties

Property names are transformed according to the "omit needless words" process described in CToSwiftNameTranslation-OmitNeedlessWords.md. This process attempts to omit superfluous type names and other words that make a name fit the Cocoa Naming Guidelines for Objective-C but not the Swift API Design Guidelines. The transformation is based on the property's type and the type of the enclosing context.

If the getter of a property overrides a superclass method or matches a method in an adopted protocol that is also a property accessor, the Swift name of the "overridden" accessor's property will be used for consistency. If there's more than one such name, one is chosen arbitrarily.

Properties with the type `BOOL` or `Boolean` use the name of the getter as the name of the Swift property by default, rather than the name of the property in Objective-C. This accounts for a difference in Swift and Objective-C naming conventions for boolean properties that use "is".

```
@property(getter=isContrivedExample) BOOL contrivedExample;
@property BOOL hasAnotherForm;
```

```
var isContrivedExample: Bool { get set }
var hasAnotherForm: Bool { get set }
```

This rule should probably have applied to C's native `bool` as well.

A property declaration with the `SwiftImportPropertyAsAccessors` API note will not be imported at all, and its accessors will be imported as methods. Additionally, properties whose names start with "accessibility" in the NSAccessibility protocol are always imported as methods, as are properties whose names start with "accessibility" in an `@interface` declaration (class or category) that provides the adoption of NSAccessibility.

Objective-C code has historically not been consistent about whether the NSAccessibility declarations should be considered properties and therefore the Swift compiler chooses to import them as methods, as a sort of lowest common denominator.

`swift_private`

The `swift_private` Clang attribute prepends `__` onto the base name of any declaration being imported except initializers. For initializers with no arguments, a dummy `Void` argument with the name `__` is inserted; otherwise, the label for the first argument has `__` prepended. This transformation takes place after any other name manipulation, unless the declaration has a custom name. It will not occur if the declaration is an override or matches a protocol requirement; in that case the name needs to match the "overridden" declaration.

```
@interface Example : NSObject
- (instancetype)initWithValue:(int)value __attribute__((swift_private));
@property(readonly) int value __attribute__((swift_private));
@end
```

```
// Usually seen as NS_REFINED_FOR_SWIFT
```

```
class Example: NSObject {
  init(__value: Int32)
  var __value: Int32 { get }
}
```

The purpose of this annotation is to allow a more idiomatic implementation to be provided in Swift. The effect of `swift_private` is inherited from an enum onto its elements if the enum is not imported as an error code enum, an `@objc` enum, or an option set.

The original intent of the `swift_private` attribute was additionally to limit access to a Swift module with the same name as the owning Clang module, e.g. the Swift half of a mixed-source framework. However, this restriction has not been implemented as of Swift 5.1.

For "historical reasons", the `swift_private` attribute is ignored on factory methods with no arguments imported as initializers. This is essentially matching the behavior of older Swift compilers for source compatibility in case someone has marked such a factory method as `swift_private`.

Custom names

The `swift_name` Clang attribute can be used to control how a declaration imports into Swift. If it's valid, the value of the `swift_name` attribute always overrides any other name transformation rules (prefix-stripping, underscore-prepending, etc.)

Types and globals

The `swift_name` attribute can be used to give a type or a global a custom name. In the simplest form, the value of the attribute must be a valid Swift identifier.

```
__attribute__((swift_name("SpacecraftCoordinates")))
struct SPKSpacecraftCoordinates {
  double x, y, z, t; // space and time, of course
};

// Usually seen as NS_SWIFT_NAME.
```

```
struct SpacecraftCoordinates {
  var x, y, z, t: Double
}
```

Import-as-member

A custom name that starts with an identifier followed by a period is taken to be a member name. The identifier should be the imported Swift name of a C/Objective-C type in the same module. In this case, the type or global will be imported as a static member of the named type.

```
__attribute__((swift_name("SpacecraftCoordinates.earth")))
extern const struct SPKSpacecraftCoordinates SPKSpacecraftCoordinatesEarth;
```

```
extension SpacecraftCoordinates {
  static var earth: SpacecraftCoordinates { get }
}
```

Note that types cannot be imported as members of protocols.

The "in the same module" restriction is considered a technical limitation; a forward declaration of the base type will work around it.

C functions with custom names

The `swift_name` attribute can be used to give a C function a custom name. The value of the attribute must be a full Swift function name, including parameter labels.

```
__attribute__((swift_name("doSomething(to:bar:)")))
void doSomethingToFoo(Foo *foo, int bar);
```

```
func doSomething(to foo: UnsafeMutablePointer<Foo>, bar: Int32)
```

An underscore can be used in place of an empty parameter label, as in Swift.

A C function with zero arguments and a non-`void` return type can also be imported as a computed variable by using the `getter:` prefix on the name. A function with one argument and a `void` return type can optionally serve as the setter for that variable using `setter:`.

```
__attribute__((swift_name("getter:globalCounter()")))
int getGlobalCounter(void);
__attribute__((swift_name("setter:globalCounter(_:)")))
void setGlobalCounter(int newValue);
```

```
var globalCounter: Int32 { get set }
```

Note that the argument lists must still be present even though the name used is the name of the variable. (Also note the `void` parameter list to specify a C function with zero arguments. This is required!)

Variables with setters and no getters are not supported.

Import-as-member

Like types and globals, functions can be imported as static members of types.

```
__attribute__((swift_name("NSSound.beep()")))
void NSBeep(void);
```

```
extension NSSound {
  static func beep()
}
```

However, by giving a parameter the label `self`, a function can also be imported as an instance member of a type **T**. In this case, the parameter labeled `self` must either have the type **T** itself, or be a pointer to **T**. The latter is only valid if **T** is imported as a value type; if the pointer is non-`const`, the resulting method will be `mutating`. If **T** is a class, the function will be `final`.

```
typedef struct {
  int value;
} Counter;

__attribute__((swift_name("Counter.printValue(self:)")))
void CounterPrintValue(Counter c);
__attribute__((swift_name("Counter.printValue2(self:)")))
void CounterPrintValue2(const Counter *c);
__attribute__((swift_name("Counter.resetValue(self:)")))
void CounterResetValue(Counter *c);
```

```
struct Counter {
  var value: Int32 { get set }
}

extension Counter {
  func printValue()
  func printValue2()
  mutating func resetValue()
}
```

This also applies to getters and setters, to be imported as instance properties.

```
__attribute__((swift_name("getter:Counter.absoluteValue(self:)")))
int CounterGetAbsoluteValue(Counter c);
```

```
extension Counter {
  var absoluteValue: Int32 { get }
}
```

Finally, functions can be imported as initializers as well by using the base name `init`. These are considered "factory" initializers and are never inherited or overridable. They must not have a `self` parameter.

```
__attribute__((swift_name("Counter.init(initialValue:)")))
Counter CounterCreateWithInitialValue(int value);
```

```
extension Counter {
  /* non-inherited */ init(initialValue value: Int32)
}
```

Enumerators (enum cases)

The `swift_name` attribute can be used to rename enumerators. As mentioned above, not only does no further prefix-stripping occur on the resulting name, but the presence of a custom name removes the enum case from the computation of a prefix for the other cases.

```
// Actual example from Apple's SDKs; in fact, the first shipping example of
// swift_name on an enumerator at all!
typedef NS_ENUM(NSUInteger, NSXMLNodeKind) {
  NSXMLInvalidKind = 0,
  NSXMLDocumentKind,
  NSXMLElementKind,
  NSXMLAttributeKind,
  NSXMLNamespaceKind,
  NSXMLProcessingInstructionKind,
  NSXMLCommentKind,
  NSXMLTextKind,
  NSXMLDTDKind NS_SWIFT_NAME(DTDKind),
  NSXMLEntityDeclarationKind,
  NSXMLAttributeDeclarationKind,
  NSXMLElementDeclarationKind,
  NSXMLNotationDeclarationKind
};
```

```
public enum Kind : UInt {
  case invalid
  case document
  case element
  case attribute
  case namespace
  case processingInstruction
  case comment
  case text
  case DTDKind
  case entityDeclaration
  case attributeDeclaration
  case elementDeclaration
  case notationDeclaration
}
```

Although enumerators always have global scope in C, they are often imported as members in Swift, and thus the `swift_name` attribute cannot be used to import them as members of another type unless the enum type is anonymous.

Currently, `swift_name` does not even allow importing an enum case as a member of the enum type itself, even if the enum is not recognized as an `@objc` enum, error code enum, or option set (i.e. the situation where a case is imported as a global constant).

Fields of structs and unions; Objective-C properties

The `swift_name` attribute can be applied to rename a struct or union field or an Objective-C property (whether on a class or a protocol). The value of the attribute must be a valid Swift identifier.

```
struct SPKSpaceflightBooking {
  const SPKLocation * _Nullable destination;
  bool roundTrip __attribute__((swift_name("isRoundTrip")));
};
```

```
struct SPKSpaceflightBooking {
  var destination: UnsafePointer<SPKLocation>?
  var isRoundTrip: Bool
}
```

Objective-C methods

The `swift_name` attribute can be used to give an Objective-C method a custom name. The value of the attribute must be a full Swift function name, including parameter labels.

```
- (void)doSomethingToFoo:(Foo *)foo bar:(int)bar
  __attribute__((swift_name("doSomethingImportant(to:bar:)")));
```

```
func doSomethingImportant(to foo: UnsafeMutablePointer<Foo>, bar: Int32)
```

As with functions, an underscore can be used to represent an empty parameter label.

Methods that follow the NSError out-parameter convention may provide one fewer parameter label than the number of parameters in the original method to indicate that a parameter should be dropped, but they do not have to. The `swift_error` attribute is still respected even when using a custom name for purposes of transforming an NSError out-parameter and the method return type.

```
- (BOOL)doSomethingRiskyAndReturnError:(NSError **)error
  __attribute__((swift_name("doSomethingRisky()")));
- (BOOL)doSomethingContrived:(NSString *)action error:(NSError **)outError
  __attribute__((swift_name("doSomethingContrived(_:error:)")));
```

```
func doSomethingRisky() throws
func doSomethingContrived(_ action: String, error: ()) throws
```

A base name of "init" can be used on a *class* method that returns `instancetype` or the containing static type in order to import that method as an initializer. Any other custom name *prevents* a class method from being imported as an initializer even if it would normally be inferred as one.

```
+ (Action *)makeActionWithHandler:(void(^)(void))handler
  __attribute__((swift_name("init(handler:)")));
```

```
+ (instancetype)makeActionWithName:(NSString *)name
  __attribute__((swift_name("init(name:)")));
```

```
/* non-inherited */ init(handler: () -> Void)
init(name: String)
```

A no-argument method imported as an initializer can be given a dummy argument label to disambiguate it from the no-argument `init()`, whether the method is an init-family instance method or a factory class method in Objective-C.

```
- (instancetype)initSafely
  __attribute__((swift_name("init(safe:)")));
+ (instancetype)makeDefaultAction
  __attribute__((swift_name("init(default:)")));
```

```
init(safe: ())
init(default: ())
```

A custom name on an instance method with one of Objective-C's subscript selectors (`objectAtIndexedSubscript:`, `objectForKeyedSubscript:`, `setObject:atIndexedSubscript:`, or `setObject:forKeyedSubscript:`) prevents that method from being imported as a subscript or used as the accessor for another subscript.

Currently, this only works if both methods in a read/write subscript are given custom names; if just one is, a read/write subscript will still be formed. A read-only subscript only has one method to rename.