Xcode Release Notes

About Xcode 6.3 beta 3

Supported Configurations

Xcode 6.3 beta 3 requires a Mac running OS X 10.10.

Xcode 6.3 beta 3 includes SDKs for iOS 8.3 and OS X versions 10.9 and 10.10. To develop apps targeting prior versions of OS X or iOS, see the section "About SDKs and the iOS Simulator" in *What's New in Xcode* available on developer.apple.com or from the Help > What's New in Xcode command when running Xcode.

Installation

This release is a single application bundle. To install, double-click the downloaded DMG file, and drag the Xcode-beta.app file to your Applications folder.

From within Xcode, you launch additional developer tools, such as Instruments and FileMerge, via the Xcode > Open Developer Tool command. You can keep the additional tools in the Dock for direct access when Xcode is not running.

Installing Xcode on OS X Server

To use Xcode's Continuous Integration service with this Xcode beta, you need OS X 10.10 with OS X Server 4.0.

Once you have installed OS X, OS X Server and Xcode, point OS X Server to this Xcode beta:

1. Open Server.app
2. Select the Xcode service
3. Choose Xcode

Technical Support and Learning Resources

Apple offers a number of resources where you can get Xcode development support:

• http://developer.apple.com: The Apple Developer website is the best source for up-to-date technical documentation on Xcode, iOS, and OS X.

• http://developer.apple.com/xcode: The Xcode home page on the Apple Developer website provides information on acquiring the latest version of Xcode.

• http://devforums.apple.com: The Apple Developer Forums are a good place to interact with fellow developers and Apple engineers, in a moderated web forum that also offers email notifications. The Developer Forums also feature a dedicated topic for Xcode developer previews.

Use http://bugreport.apple.com to report issues to Apple. Include detailed information of the issue, including the system and developer tools version information, and any relevant crash logs or console messages.

New Features in Xcode 6.3

App Store and Test Flight Crash Logs Organizer

Xcode 6.3 includes a new feature to help opted-in App Store users and TestFlight users to collect and analyze crash log data for your apps.

Xcode Playgrounds Enhancements

Playgrounds have been enhanced with documentation authoring using inline marked-up comments, inline playground results, the ability to view and edit resources embedded in playgrounds, and the ability to integrate auxiliary source files into Playgrounds. These features enable the creation of rich new experiences in playgrounds.

Swift 1.2

Xcode 6.3 includes a new version of the Swift language. It includes a number of noteworthy changes to the language, detailed in the *New in Xcode 6.3 beta* section. Xcode 6.3 provides a migrator for moving your code to Swift 1.2.

Objective-C

Objective-C code can now directly express the nullability of pointers in header files, improving interoperability between Swift and Objective-C code.

Apple LLVM Compiler Version 6.1

Xcode 6.3 updates the Apple LLVM compiler to version 6.1, including support C++14, enhanced warning diagnostics, and new optimizations.

ARM64 Intrinsics

The argument ordering for the arm64 vfma/vfms lane intrinsics has changed. See "ARM64 Intrinsics Changes" for details.

New in Xcode 6.3 beta 3

Force Touch Support

- Xcode uses Force Touch trackpad for Macs that include it, and supports configuring Force Touch trackpad functionality in Xcode's Interface Builder editor for NSButton and NSSegmentedControl. Adopting Force Touch in Interface Builder requires OS X Yosemite 10.10.3 (14D98g). (16140561, 16140600, 18660545)

Xcode Playground Enhancements

- Playgrounds can now be upgraded to the new format by selecting the Editor > Upgrade Playground… menu item. The rich comments and supporting source files features are only supported in Playgrounds created with Xcode 6.3 or later. (19938996)

- Playgrounds now expose their structure in the Project navigator. To show the Project navigator, select View > Navigators > Show Project Navigator. This allows you to use resources (e.g. images) from within your Playground: twist open the Playground to see the Resources folder and drag them in. (19115173)

- Playgrounds now let you provide auxiliary support source files, which are compiled into a module and automatically imported into your Playground. To use the new supporting source files feature, twist open the playground in the project navigator to see the new Sources folder, which has a single file named "SupportCode.swift" by default. Add code to that file, or create new source files in this folder, which will all be automatically compiled into a module and automatically imported into your playground. (19460887)

Swift Standard Library

• `flatMap` was added to the standard library. `flatMap` is the function that maps a function over something and returns the result flattened one level. FlatMap has many uses, such as to flatten an array: `[[1,2], [3,4]].flatMap { $0 }` or to chain optionals with functions: `[[1,2], [3,4]].first.flatMap { find($0, 1) }`. (19881534)

Debugger Enhancements

LLDB's Objective-C expression parser can now import modules. Any subsequent expression can rely on function and method prototypes defined in the module:

```
(lldb) p @import Foundation
(lldb) p NSPointFromString(@"{10.0, 20.0}");
(NSPoint) $1 = (x = 10, y = 20)
```

Before Xcode 6.3, methods and functions without debug information required explicit typecasts to specify their return type. Importing modules allows a developer to avoid the more labor-intensive process of determining and specifying this information manually:

```
(lldb) p NSPointFromString(@"{10.0, 20.0}");
error: 'NSPointFromString' has unknown return type; cast the call to its declared return type
error: 1 errors parsing expression
```

```
(lldb) p (NSPoint)NSPointFromString(@"{10.0, 20.0}”);
(NSPoint) $0 = (x = 10, y = 20)
```

Other benefits of importing modules include better error messages, access to variadic functions when running on 64-bit devices, and eliminating potentially incorrect inferred argument types. In several cases, not having a proper function prototype could lead to unexpected failures. (18782288)

Issues Resolved in Xcode 6.3 beta 3

Playgrounds

- Using the "/*: */ multi-line variant of Playgrounds rich comment markers no longer results in the comment block being duplicated further down in the document. (19917362)

Swift Language

- The compiler no longer generates invalid code on casting of NSArray to Swift array in Release mode (19724405)

- Warnings are now emitted for uses of APIs that are deprecated on all deployment targets. (17406050)

- Bridging an empty Swift array from Objective-C back into Swift no longer causes a segfault. (19883644)

- Providing a stored property with an `@objc` attribute that has a name now affects the selector names of the property's getters and setters. (19963219) For example:

```
class Foo : NSObject {
   @objc(label) var title: String
   // Objective-C getter will be named “label”
   // Objective-C setter will be named “setLabel:”
 }
```

- A previous beta began requiring `self.` on the use of instance members within local functions (as it is for closures). This requirement was considered too broad and has been removed. (19945738)

Compiler

- When building a module, compilations that import that module no longer give "error: timed out waiting to acquire lock file for module" instead of the correct error message. (19840043)

iOS Simulator

- iOS Simulator has been fixed and now maintains network connectivity when the host's network configuration changes. (17867038)

Known Issues in Xcode 6.3 beta 3

Playgrounds

- Description: Sometimes Xcode will crash after editing a supporting source file for a playground and then viewing the playground itself. This happens when using the Project navigator to move back and forth between the playground and its enclosed files. (20094959)

  Workaround: Open and edit the playground supporting source files in their own windows

- Description: Playgrounds with supporting source files sometimes fail to execute, showing an error in the Console Output like: “Playground execution failed: error: Couldn’t lookup symbols: __TF16EnrichMe_Sources2hiFT_T_ __TF16EnrichMe_Sourcesau2piSd”. (20100043)

  Workaround: View the supporting source file, then view and re-execute the playground

- Description: Playgrounds with supporting source files sometimes fail to execute after changing the name of a function in the support file. You may see an error in the Console Output like: "Playground execution failed: MyPlayground.playground:6:1: error: use of unresolved identifier ‘myFunc’". (20109247)

  Workaround: Quit and relaunch Xcode. You may then need to view the supporting file, then view and re-execute the playground

Swift Language

- Swift 1.2 is stricter about checking type-based overloading of `@objc` methods and initializers which is something not supported by Objective-C. (19826275)

```
// Has the Objective-C selector "performOperation:".
func performOperation(op: NSOperation) { /* do something */ }
// Also has the selector "performOperation:".
func performOperation(fn: () -> Void) {
     self.performOperation(NSBlockOperation(block: fn))
}
```

This code would work when invoked from Swift, but could easily crash if invoked from Objective-C.

Workaround: Use a type that is not supported by Objective-C to prevent the Swift compiler from exposing the member to the Objective-C runtime by doing one of the following:

- If it makes sense, mark the member as `private` to disable inference of `@objc`.
- Rename the corresponding Objective-C method with the `@objc` attribute. For example:
```
@objc(performOperationWithBlock:)
func performOperation(fn: () -> Void) { … }
```
- Otherwise, use a dummy parameter with a default value: `_ nonobjc: () = ()`.

- Overrides of methods exposed to Objective-C in private subclasses are not inferred to be `@objc`, causing the Swift compiler to crash. (19935352)

Workaround: Explicitly add the `@objc` attribute to any such overriding methods.

- When subclassing UITableViewController, if you try to create your own initializer you will see an error telling you "initializer does not override a designated initializer from its superclass." (19775924)

  Workaround: To override the designated initializer initWithNibName:bundle: you will need to declare it as a designated initializer in a class extension in an Objective-C bridging header. The following steps will guide you through this process:

  1. In your Swift project, create a new empty iOS Objective-C file. This will trigger a sheet asking you "Would you like to configure an Objective-C bridging header?"
  2. Tap "Yes" to create a bridging header
  3. Inside [YOURPROJECTNAME]-Bridging-Header.h add the following code:

     ```
     @import UIKit;

     @interface UITableViewController() // Extend UITableViewController to
     work around 19775924
     - (instancetype)initWithNibName:(NSString *)nibNameOrNil bundle:
     (NSBundle *)nibBundleOrNil NS_DESIGNATED_INITIALIZER ;
     @end
     ```

  4. Rebuild your project

Xcode Run and Debug

- Localization settings made in a target's scheme (as well as other settings made on the command line) will not be honored on launch in the iOS Simulator with iOS 8 runtimes. (19490124)

  Workaround: Choose the desired language in the Settings app.

Migration Assistant

- “Convert to Latest Swift” may generate build errors when run. These errors can be safely ignored and don’t affect the source changes that are produced. (19650497)

iOS Simulator

- Under some circumstances, iOS extensions will need to be manually enabled before you can debug them. (18603937)

- When launching a WatchKit app on the iOS Simulator, the Watch Simulator may not automatically launch. (19895243)

  Workaround: Launch the Watch Simulator before launching the WatchKit app on the iOS Simulator.

New in Xcode 6.3 beta 2

App Store and Test Flight Crash Logs Organizer

- Xcode can now display crash reports gathered from opted-in App Store users and TestFlight users. To view crash reports for your apps, first enter your developer accounts into Xcode's Accounts pane. View crash reports for the iOS apps associated with your developer accounts within Xcode's Organizer window. Crash reports are only available for apps that were uploaded to iTunes Connect with symbol information. Xcode provides a list of the top crashes for each of your apps, and the crash reports are fully symbolicated and aggregated on Apple's servers. Xcode also provides workflows for managing your crash reports and viewing backtraces directly beside your project's source code. (14995491)

  For more information, see the Crashes Organizer Help in the Xcode documentation.

Xcode Playgrounds Enhancements

- Playgrounds now offer an easy way to create and edit rich documentation using marked-up text. Use the new "//:" or "/*: */" style comment to indicate when text should be shown as a rich comment. Change the viewing mode of a Playground by using the "Show Documentation as Rich Text" and "Show Documentation as Raw Text" menu items in the Editor menu. For more information, see Playground Markup Format Reference in the Xcode documentation. (19265300)

- Playground results are now shown inline, rather than in the timeline view. When there are multiple results on a line, you can toggle between viewing a single result and a listing of all the results. For result sets that are numbers, there is the added option of viewing as a graph. Results can be resized to show more or less information. For more information, see Exploring and Evaluating Swift Code in a Playground in the Xcode documentation. (19259877)

- Playground scrolling and performance has been improved.

Swift Performance

- A new compilation mode has been introduced for Swift called Whole Module Optimization. This option optimizes all of the files in a target together and enables better performance (at the cost of increased compile time). The new flag can be enabled in Xcode using the "Whole Module Optimization" build setting or by using the swiftc command line tool with the flag "`–whole–module–optimization`". (18603795)

Swift Language Enhancements

- The "if-let" syntax has been extended to support a single leading boolean condition along with optional binding let clauses. (19797158)

  For example:
  ```
  if someValue > 42 && someOtherThing < 19,
       let a = getOptionalThing() where a > someValue {
  }
  ```

- The `@autoclosure` attribute has a second form, `@autoclosure(escaping)` that provides the same caller-side syntax as `@autoclosure` but allows the resulting closure to escape in the implementation. (19499207)
  For example:

```
func lazyAssertion(@autoclosure(escaping) condition: () -> Bool,
                   message: String = "") {
lazyAssertions.append(condition) // escapes
}
lazyAssertion(1 == 2, message: "fail eventually")
```

Swift Language Changes

- The precedence of the Nil Coalescing Operator ("`??`") has been raised to bind tighter than short-circuiting logical and comparison operators, but looser than `as` conversions and `range` operators. This provides more useful behavior for expressions like:

```
if allowEmpty || items?.count ?? 0 > 0 {
```

- The "`&/`" and "&%" operators were removed, to simplify the language and improve consistency. Unlike the "&+", "&–", and "&*" operators, these did not provide two's-complement arithmetic behavior — they provided special case behavior for division/remainder by zero and Int.min/-1. We prefer that these tests be written explicitly in the code as comparisons if needed. (17926954).

- Constructing a `UInt8` from an ASCII value now requires the `ascii` keyword parameter. Using non-ASCII unicode scalars will cause this initializer to trap. (18509195)

- The C `size_t` family of types are now imported into Swift as `Int`, since Swift prefers sizes and counts to be represented as signed numbers, even if they are non-negative. This decreases the amount of explicit type conversion between `Int` and `UInt`, better aligns with sizeof returning Int, and provides safer arithmetic properties. (18949559)

- Classes that do not inherit from NSObject but do adopt an @objc protocol will need to explicitly mark those methods, properties, and initializers used to satisfy the protocol requirements as @objc.

  For example:

```
@objc protocol SomethingDelegate {
     func didSomething()
}

class MySomethingDelegate : SomethingDelegate {
     @objc func didSomething() { … }
}
```

Swift Standard Library Enhancements

- The function `zip` was added. It joins two sequences together into one sequence of tuples. (17292393)

- `utf16Count` is removed from `String`. Instead use count on the UTF16 view of the String. (17627758)

For example:

```
    count(string.utf16).
```

Debugger Enhancements

- Evaluating Swift expressions performance is improved especially when debugging code running on devices. This will be most noticeable in the Swift REPL and when issuing LLDB commands such as `p`, `po`, and `expression`. (19213054)

- Significant improvements in LLDB's Swift support have addressed many known issues with the Swift debugging experience. (19656017)

Issues Resolved in Xcode 6.3 beta 2

Xcode Interface Builder

- The Interface Builder editor now supports connecting between source code and a UIRefreshControl. (17935413)
- The UIVisualEffectView item is now available in the object library. (19779350)

Xcode Swift Support

- Addressed more instances of intermittent SourceKitService crashes while browsing a project. (19059478)
- Several common issues affecting the “Convert to Latest Swift” tool have been fixed.

Xcode Comparison View

- Comparison View’s vertical scrollbar now works when viewing added files. (19720720)

Swift Compiler

- `@objc` enums no longer cause the compiler to crash when used from another file. (19775284)
- Fixed a use after free crash in `lowercaseString` and `uppercaseString`. (19801253)
- Static stored properties can now be used in classes marked @objc or subclass NSObject, and are accessible from Objective-C. (19784053)
- This expression now parses correctly, without need for parentheses:

```
dict[someKey] as? Int ?? 5
```

New in Xcode 6.3 beta

Swift Migrator from Swift 1.1 to Swift 1.2

- A source migrator tool has been provided to help migrate your Swift source code from Swift 1.1 (Xcode 6.1.1) to Swift 1.2 (Xcode 6.3.) In Xcode, select `Edit > Convert > To Swift 1.2`.

Swift Language Enhancements

- Swift now supports building targets incrementally, i.e. not rebuilding every Swift source file in a target when a single file is changed. This is based on a conservative dependency analysis, so you may still see more files rebuilding than absolutely necessary. If you find any cases where a file is not rebuilt when it should be, please file a bug report; running Clean on your target should then allow you to complete your build normally. (18248514)

- A new `Set` data structure is included which provides a generic collection of unique elements with full value semantics. It bridges with `NSSet`, providing functionality analogous to `Array` and `Dictionary`. (14661754)

- The "`if let`" construct has been expanded to allow testing multiple optionals and guarding conditions in a single if (or while) statement using syntax similar to generic constraints:

```
if let a = foo(), b = bar() where a < b,
   let c = baz() {
}
```

  This allows you to test multiple optionals and include intervening boolean conditions, without introducing undesirable nesting (i.e., to avoid the optional unwrapping "pyramid of doom"). (19382942)

- `let` constants have been generalized to no longer require immediate initialization. The new rule is that a `let` constant must be initialized before use (like a `var`), and that it may only be initialized: not reassigned or mutated after initialization. This enables patterns like:

```
let x: SomeThing
if condition {
  x = foo()
} else {
  x = bar()
}
use(x)
```

  which formerly required the use of a `var`, even though there is no mutation taking place. (16181314)

- "`static`" methods and properties are now allowed in classes (as an alias for "`class final`"). You are now allowed to declare `static` stored properties in classes, which have global storage and are lazily initialized on first access (like global variables). Protocols now declare type requirements as "`static`" requirements instead of declaring them as "`class`" requirements. (17198298)

- Type inference for single-expression closures has been improved in several ways:
  - Closures that are comprised of a single return statement are now type checked as single-expression closures.
  - Unannotated single-expression closures with non-`Void` return types can now be used in `Void` contexts.
  - Situations where a multi-statement closure's type could not be inferred because of a missing return-type annotation are now properly diagnosed.

- Swift enums can now be exported to Objective-C using the `@objc` attribute. `@objc` enums must declare an integer raw type, and cannot be generic or use associated values. Because Objective-C enums are not namespaced, enum cases are imported into Objective-C as the concatenation of the enum name and case name. (16967385)

  For example, this Swift declaration:

  ```
  @objc
  enum Bear: Int {
     case Black, Grizzly, Polar
  }
  ```

  imports into Objective-C as:

  ```
  typedef NS_ENUM(NSInteger, Bear) {
     BearBlack, BearGrizzly, BearPolar
  };
  ```

- Objective-C language extensions are now available to indicate the nullability of pointers and blocks in Objective-C APIs, allowing your Objective-C APIs to be imported without `ImplicitlyUnwrappedOptional`. See below for more details (18868820).

- Swift can now partially import C aggregates containing unions, bitfields, SIMD vector types, and other C language features that are not natively supported in Swift. The unsupported fields will not be accessible from Swift, but C and Objective-C APIs that have arguments and return values of these types can be used in Swift. This includes the Foundation `NSDecimal` type and the GLKit `GLKVector` and `GLKMatrix` types, among others. (15951448)

- Imported C structs now have a default initializer in Swift, which initializes all of the struct's fields to zero. For example:

  ```
  import Darwin
  var devNullStat = stat()
  stat("/dev/null", &devNullStat)
  ```

  If a structure contains fields that cannot be correctly zero initialized (i.e. pointer fields marked with the new `__nonnull` modifier), this default initializer will be suppressed. (18338802)

- New APIs for converting among the Index types for `String`, `String.UnicodeScalarView`, `String.UTF16View`, and `String.UTF8View` are available, as well as APIs for converting each of the String views into Strings. (18018911)

- Type values now print as the full demangled type name when used with println or string interpolation. (18947381)

```
toString(Int.self)          // prints “Swift.Int"
println([Float].self)       // prints "Swift.Array<Swift.Float>”
println((Int, String).self) // prints "(Swift.Int, Swift.String)"
```

- A new “@noescape” attribute may be used on closure parameters to functions. This indicates that the parameter is only ever called (or passed as an @noescape parameter in a call), which means that it cannot outlive the lifetime of the call. This enables some minor performance optimizations, but more importantly disables the “self.” requirement in closure arguments. This enables control-flow-like functions to be more transparent about their behavior. In a future beta, the standard library will adopt this attribute in functions like `autoreleasepool()`. (16323038)

```
func autoreleasepool(@noescape code: () -> ()) {
  pushAutoreleasePool()
  code()
  popAutoreleasePool()
}
```

- Performance is substantially improved over Swift 1.1 in many cases. For example, multidimensional arrays are algorithmically faster in some cases, unoptimized code is much faster in many cases, and many other improvements have been made.

- The diagnostics emitted for expression type check errors are greatly improved in many cases. (18869019)

- Type checker performance for many common expression kinds has been greatly improved. This can significantly improve build times and reduces the number of “expression too complex” errors. (18868985)

Swift Language Changes

- The notions of guaranteed conversion and “forced failable” conversion are now separated into two operators. Forced failable conversion now uses the `as!` operator. The `!` makes it clear to readers of code that the cast may fail and produce a runtime error. The “`as`” operator remains for upcasts (e.g. “someDerivedValue as Base”) and type annotations (“0 as Int8”) which are guaranteed to never fail.(19031957)

- Immutable (`let`) properties in struct and class initializers have been revised to standardize on a general “`lets` are singly initialized but never reassigned or mutated” model. Previously, they were completely mutable within the body of initializers. Now, they are only allowed to be assigned to once to provide their value. If the property has an initial value in its declaration, that counts as the initial value for all initializers. (19035287)

- The implicit conversions from bridged Objective-C classes (NSString/NSArray/NSDictionary) to their corresponding Swift value types (String/Array/Dictionary) have been removed, making the Swift type system simpler and more predictable. (18311362)

  This means that the following code will no longer work:

  ```
  import Foundation
  func log(s: String) { println(x) }
  let ns: NSString = "some NSString" // okay: literals still work
  log(ns)   // fails with the error
            // "'NSString' is not convertible to 'String'"
  ```

  In order to perform such a bridging conversion, make the conversion explicit with the as keyword:

  ```
  log(ns as String) // succeeds
  ```

  Implicit conversions from Swift value types to their bridged Objective-C classes are still permitted. For example:

  ```
  func nsLog(ns: NSString) { println(ns) }
  let s: String = “some String”
  nsLog(s) // okay: implicit conversion from String to NSString is permitted
  ```

  Note that these Cocoa types in Objective-C headers are still automatically bridged to their corresponding Swift type, which means that code is only affected if it is explicitly referencing (e.g.) NSString in a Swift source file. We recommend using the corresponding Swift types (e.g. String) directly unless you are doing something advanced, like implementing a subclass in the class cluster.

- The @autoclosure attribute is now an attribute on a parameter, not an attribute on the parameter’s type. (15217242)

  Where before you might have used:

  ```
  func assert(predicate : @autoclosure () -> Bool) {… }
  ```

  you now write this as:

  ```
  func assert(@autoclosure predicate : () -> Bool) {… }
  ```

- The @autoclosure attribute on parameters now implies the new @noescape attribute.

- Curried function parameters can now specify argument labels (17237268):

  ```
  func curryUnnamed(a: Int)(_ b: Int) { return a + b }
  curryUnnamed(1)(2)

  func curryNamed(first a: Int)(second b: Int) -> Int { return a + b }
  curryNamed(first: 1)(second: 2)
  ```

- Swift now detects discrepancies between overloading and overriding in the Swift type system and the effective behavior seen via the Objective-C runtime. (18391046, 18383574)

  For example, the following conflict between the Objective-C setter for “property” in a class and the method “setProperty” in its extension is now diagnosed:

```
class A : NSObject {
  var property: String = "Hello" // note: Objective-C method 'setProperty:’
                                // previously declared by setter for
                                // 'property’ here
}

extension A {
  func setProperty(str: String) { }   // error: method ‘setProperty’
                                     // redeclares Objective-C method
                                     //'setProperty:’
}
```

  Similar checking applies to accidental overrides in the Objective-C runtime:

```
class B : NSObject {
  func method(arg: String) { }   // note: overridden declaration
                                // here has type ‘(String) -> ()’
}

class C : B {
  func method(arg: [String]) { } // error: overriding method with
                                // selector ‘method:’ has incompatible
                                // type ‘([String]) -> ()’
}
```

  as well as protocol conformances:

```
class MyDelegate : NSObject, NSURLSessionDelegate {
  func URLSession(session: NSURLSession, didBecomeInvalidWithError: Bool){ }
     // error: Objective-C method 'URLSession:didBecomeInvalidWithError:'
     // provided by method 'URLSession(_:didBecomeInvalidWithError:)'
     // conflicts with optional requirement method
     // 'URLSession(_:didBecomeInvalidWithError:)' in protocol
     // 'NSURLSessionDelegate'
}
```

Swift Language Fixes

- Dynamic casts (“`as!`", “`as?`" and “`is`”) now work with Swift protocol types, so long as they have no associated types. (18869156)

- Adding conformances within a Playground now works as expected, for example:

```
struct Point {
  var x, y: Double
}

extension Point : Printable {
  var description: String {
    return "(\(x), \(y))"
  }
}

var p1 = Point(x: 1.5, y: 2.5)
println(p1) // prints "(1.5, 2.5)”
```

- Imported `NS_ENUM` types with undocumented values, such as `UIViewAnimationCurve`, can now be converted from their raw integer values using the `init(rawValue:)` initializer without being reset to nil. Code that used `unsafeBitCast` as a workaround for this issue can be written to use the raw value initializer. (19005771)

  For example:

```
    let animationCurve =
unsafeBitCast(userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue,
                     UIViewAnimationCurve.self)
```

  can now be written instead as:

```
    let animationCurve = UIViewAnimationCurve(rawValue:
          userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue)!
```

- Negative floating-point literals are now accepted as raw values in enums. (16504472)

- Unowned references to Objective-C objects, or Swift objects inheriting from Objective-C objects, no longer cause a crash if the object holding the unowned reference is deallocated after the referenced object has been released. (18091547)

- Variables and properties with observing accessors no longer require an explicit type if it can be inferred from the initial value expression. (18148072)

- Generic curried functions no longer produce random results when fully applied. (18988428)

- Comparing the result of a failed `NSClassFromString` lookup against nil now behaves correctly. (19318533)

- Subclasses that override base class methods with co- or contravariance in Optional types no longer cause crashes at runtime. (19321484)

  For example:

```
class Base {
  func foo(x: String) -> String? { return x }
}
class Derived: Base {
  override func foo(x: String?) -> String { return x! }
}
```

Objective-C Language Enhancements

- Objective-C APIs can now express the "nullability" of parameters, return types, properties, variables, etc. For example, here is the expression of nullability for several UITableView APIs:

```
    -(void)registerNib:(nonnull UINib *)nib forCellReuseIdentifier:(nonnull
NSString *)identifier;
     -(nullable UITableViewCell *)cellForRowAtIndexPath:(nonnull
NSIndexPath)indexPath;
    @property (nonatomic, readwrite, retain, nullable) UIView *backgroundView;
```

  The nullability qualifiers affect the optionality of the Objective-C APIs when in Swift. Instead of being imported as implicitly-unwrapped optionals (e.g., `UINib!`), `nonnull`-qualified types are imported as non-optional (e.g., `UINib`) and `nullable`-qualified types are imported as optional (e.g., `UITableViewCell?`), so the above APIs will be seen in Swift as:

```
func registerNib(nib: UINib, forCellReuseIdentifier identifier: String)
func cellForRowAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell?
var backgroundView: UIView?
```

  Nullability qualifiers can also be applied to arbitrary pointer types, including C pointers, block pointers, and C++ member pointers, using double-underscored versions of the nullability qualifiers. For example, consider a C API such as:

```
void enumerateStrings(__nonnull CFStringRef (^ __nullable callback)(void));
```

  Here, the callback itself is nullable and the result type of that callback is nonnull. This API will be usable from Swift as:

```
func enumerateStrings(callback: (() -> CFString)?)
```

  In all, there are three different kinds of nullability specifiers, which can be spelled with a double-underscore (on any pointer type) or without (for Objective-C properties, method result types, and method parameter types):

Type qualifier spelling	Objective-C property/method spelling	Swift view	Meaning
`__nonnull`	`nonnull`	Non-optional, e.g., `UINib`	The value is never expected to be nil (except perhaps due to messaging nil in the argument)
`__nullable`	`nullable`	Optional, e.g., `UITableViewCell?`	The value can be nil
`__null_unspecified`	`null_unspecified`	Implicitly-unwrapped optional, e.g., `NSDate!`	It is unknown whether the value can be nil (very rare)

Particularly in Objective-C APIs, many pointers tend to be `nonnull`. Therefore, Objective-C provides "audited" regions (via a new `#pragma`) that assume that unannotated pointers are `nonnull`. For example, the following example is equivalent to the first example, but uses audited regions to simplify the presentation:

```
    #pragma clang assume_nonnull begin
    // …
    -(void)registerNib:(UINib *)nib forCellReuseIdentifier:(NSString
*)identifier;
     -(nullable UITableViewCell *)cellForRowAtIndexPath:(NSIndexPath)indexPath;
    @property (nonatomic, readwrite, retain, nullable) UIView *backgroundView;
    // …
    #pragma clang assume_nonnull end
```

For consistency, we recommend using audited regions in all Objective-C headers that describe the nullability of their APIs, and to avoid `null_unspecified` except as a transitional tool while introducing nullability into existing headers.

Adding nullability annotations to Objective-C APIs does not affect backward compatibility or the way in which the compiler generates code. For example, `nonnull` pointers can still end up being nil in some cases, such as when messaging a nil receiver. However, nullability annotations—in addition to improving the experience in Swift—provide new warnings in Objective-C if, for example, a nil argument is passed to a `nonnull` parameter, making Objective-C APIs more expressive and easier to use correctly. (18868820)

- Objective-C APIs can now express the nullability of properties whose setters allow nil (to "reset" the value to some default) but whose getters never produce nil (because they provide some default instead) using the `null_resettable` property attribute. One such property is UIView's `tintColor`, which substitutes a default system tint color when no tint color has been specified. (19051334)

  For example:

  ```
      @property (nonatomic, retain, null_resettable) UIColor *tintColor;
  ```

  Such APIs are imported into Swift via implicitly-unwrapped optionals, e.g.,

  ```
      var tintColor: UIColor!
  ```

- Parameters of C pointer type or block pointer type can be annotated with the new `noescape` attribute to indicate that pointer argument won't "escape" the function or method it is being passed to. (19389222)

  In such cases, it's safe to (for example) pass the address of a local variable. `noescape` block pointer parameters will be imported into Swift as `@noescape` parameters:

  ```
  void executeImmediately(__attribute__((noescape)) void (^callback)(void));
  ```

  is imported into Swift as:

  ```
  func executeImmediately(@noescape callback: () -> Void)
  ```

Debugger Enhancements

- LLDB now includes a prototype for `printf()` by default when evaluating C/C++/Objective-C expressions. This improves the expression evaluation experience on arm64 devices, but may conflict with user-defined expression prefixes in .lldbinit that have a conflicting declaration of `printf()`. If you see errors during expression evaluation this may be the root cause. (19024779)

Apple LLVM Compiler Version 6.1

- Xcode 6.3 updates the Apple LLVM compiler to version 6.1.0. This new compiler includes full support for the C++14 language standard, a wide range of enhanced warning diagnostics, and new optimizations. Support for the arm64 architecture has been significantly revised to align with ARM's implementation, where the most visible impact is that a few of the vector intrinsics have changed to match ARM's specifications.

ARM64 Intrinsics Changes

- The argument ordering for the arm64 vfma/vfms lane intrinsics has changed. Since that change may not trigger compile-time errors but will break code at runtime, we are staging the transition to make it less risky. By default, the compiler now warns about any use of those intrinsics and will retain the old behavior. As soon as possible, adopt the new behavior and define the `USE_CORRECT_VFMA_INTRINSICS` macro to 1 to inform the compiler of that. You can define that macro to 0 to silence the warnings and keep the old behavior, but do not leave your code in that state for long, since support for the old behavior will be removed in a future release. (17964959)

Issues Resolved in Xcode 6.3 beta

Xcode Interface Builder

- Views that have autoresizing masks and lie inside of UITableView, UICollectionView, or NSScrollView no longer get misaligned when opening the document. (18404033)