orphan:

Constructors and Initialization in Swift

Warning

This proposal was rejected, though it helped in the design of the final Swift 1 initialization model.

Contents

Initialization in Objective-C

In Objective-C, object allocation and initialization are separate operations that are (by convention) always used together. One sends the `alloc` message to the class to allocate it, then sends an `init` (or other message in the init family) to the newly-allocated object, for example:

```
[[NSString alloc] initWithUTF8String:"initialization"]
```

Two-Phase Initialization

The separation of allocation and initialization implies that initialization is a two-phase process. In the first phase (allocation), memory is allocated and the memory associated with all instance variables is zero'd out. In the second phase, in which the appropriate "init" instance method is invoked, the instance variables are (manually) initialized.

Note that an object is considered to be fully constructor after allocation but before initialization. Therefore, a message send initiated from a superclass's `init` can invoke a method in a subclass whose own `init` has not completed. A contrived example:

```
@interface A : NSObject {
  NSString *_description;
}
- (id)init;
- (NSString*)description;
@end

@implementation A
- (id)init {
  self = [super init]
  if (self) {
    _description = [self description];
  }
  return self;
}

- (NSString *)description {
  return @"A";
}
@end

@interface B : A
@property NSString *title;
@end

@implementation B
- (id)init {
  self = [super init]
  if (self) {
    self->title = @"Hello";
  }
  return self;
}

-(NSString *)description {
  return self->title;
}
@end
```

During the second phase of initialization, A's `-init` method invokes the `-description` method, which ends up in B's `-description`. Here, `title` will be `nil` even though the apparent invariant (when looking at B's `-init` method) is that `title` is never nil, because B's `-init` method has not yet finished execution.

In a language with single-phase initialization (such as C++, Java, or C# constructors), the object is considered to have the type of the constructor currently executing for the purposes of dynamic dispatch. For the Objective-C example above, this would mean that when A's `-init` sends the `description` message, it would invoke A's `-description`. This is somewhat safer than two-phase initialization, because the programmer does not have to deal with the possibility of executing one's methods before the initialization of one's instance variables have completed. It is also less flexible.

Designated Initializers

One of the benefits of Objective-C's `init` methods is that they are instance methods, and therefore are inherited. One need not override the `init` methods in a subclass unless that subclass has additional instance variables that require initialization. However, when a subclass does introduce additional instance variables that require initialization, which `init` methods should it override? If the answer is "all of them", then initializer inheritance isn't all that useful.

Objective-C convention has the notion of a designated initializer which is, roughly, an initializer method that is responsible for calling one of it's superclass's initializers, then initializing its own instance variables. Initializers that are not designated initializers are "secondary" initializers: they typically delegate to another initializer (eventually terminating the chain at a designated initializer) rather than performing initialization themselves. For example, consider the following `Task` class (from the aforementioned "Concepts in Objective-C Programming" document):

```
@interface Task
@property NSString *title;
@property NSDate *date;

- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
- (id)initWithTitle:(NSString *)aTitle;
- (id)init;
@end

@implementation Task
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate {
  title = aTitle;
  date = aDate;
  return self;
}

- (id)initWithTitle:(NSString *)aTitle {
  return [self initWithTitle:aTitle date:[NSDate date]];
}

- (id)init {
  return [self initWithTitle:@"Task"];
}
@end
```

The first initializer is the designated initializer, which directly initializes the instance variables from its parameters. The second two initializers are secondary initializers, which delegate to other initializers, eventually reaching the designated initializer.

A subclass should override all of its superclass's designated initializers, but it need not override the secondary initializers. We can illustrate this with a subclass of `Task` that introduces a new instance variable:

```
@interface PackagedTask : Task
@property dispatch_queue_t queue;

- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate queue:(dispatch_queue_t)aQueue;
@end

@implementation PackagedTask
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate {
  return [self initWithTitle:aTitle
               date:aDate
               queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)];
}

- (id)initWithTitle:(NSString * aTitle date:(NSDate *)aDate queue:(dispatch_queue_t)aQueue {
  self = [super initWithTitle:aTitle date:aDate];
  if (self) {
    queue = aQueue;
  }
  return self;
}
@end
```

`PackagedTask` overrides `Task`'s designated initializer, `-initWithTitle:date:`, which then becomes a secondary initializer of `PackagedTask`, whose only designated initializer is `initWithTitle:date:queue:`. The latter method invokes its superclass's designated initializer (`-[Task initWithTitle:date:]`), then initializes its own instance variable. By following the rules of designated initializers mentioned above, one ensures that the inherited secondary initializers still work. Consider the execution of `[[PackagedTask alloc] init]`:

- A `PackagedTask` object is allocated.
- `-[Task init]` executes, which delegates to `-[Task initWithTitle]`.
- `-[Task initWithTitle]` delegates to `-initWithTitle:date:`, which executes `-[PackagedTask initWithTitle:date:]`.
- `-[PackagedTask initWithTitle:date:]` invokes `-[Task initWithTitle:date:]` to initialize `Task`'s instance variables, then initializes its own instance variables.

The Middle Phase of Initialization

Objective-C's two-phase initialization actually has a third part, which occurs between the zeroing of the instance variables and the call to the initialization. This initialization invokes the default constructors for instance variables of C++ class type when those default constructors are not trivial. For example:

```
@interface A
struct X {
  X() { printf("X()\n"); }
};

@interface A : NSObject {
  X x;
}
@end
```

When constructing an object with `[[A alloc] init]`, the default constructor for `X` will execute after the instance variables are zeroed but before `+alloc` returns.

Swift Constructors

Swift's constructors merge both allocation and initialization into a single function. One constructs a new object with type construction syntax as follows:

```
Task(title:"My task", date:NSDate())
```

The object will be allocated (via Swift's allocation routines) and the corresponding constructor will be invoked to perform the initialization. The constructor itself might look like this:

```
class Task {
  var title : String
  var date : NSDate

  constructor(title : String = "Task", date : NSDate = NSDate()) {
    self.title = title
    self.date = date
  }
}
```

Due to the use of default arguments, one can create a task an any of the following ways:

```
Task()
Task(title:"My task")
Task(date:NSDate())
Task(title:"My task", date:NSDate())
```

Constructor Delegation

Although our use of default arguments has eliminated the need for the various secondary constructors in the Objective-C version of the `Task` class, there are other reasons why one might want to have one constructor in a class call another to perform initialization (called constructor *delegation*). For example, default arguments cannot make use of other argument values, and one may have a more complicated default value. For example, the default title could depend on the provided date. Swift should support constructor delegation for this use case:

```
constructor(title : String, date : NSDate = NSDate()) {
  self.title = title
  self.date = date
}

constructor(date : NSDate = NSDate()) {
  /*self.*/constructor(title:"Task created on " + date.description(),
                       date:date)
}
```

A constructor that delegates to another constructor must do so before using or initializing any of its instance variables. This property can be verified via definite initialization analysis.

Superclass Constructors

When one class inherits another, each constructor within the subclass must call one of its superclass's constructors before using or initializing any of its instance variables, which is also verified via definite initialization analysis. For example, the Swift `PackagedTask` class could be implemented as:

```
class PackagedTask : Task {
  var queue : dispatch_queue_t

  constructor(title : String, date : NSDate = NSDate(),
              queue : dispatch_queue_t = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0))
    super.constructor(title:title, date:date)
    self.queue = queue
  }
}
```

Instance Variable Initialization

Swift allows instance variables to be provided with an initializer, which will be called as part of initialization of an object. For example, we could provide initializers for the members of `Task` as follows:

```
class Task {
  var title : String = "Title"
  var date : NSDate = NSDate()
}
```

Here, one does not need to write any constructor: a default, zero-parameter constructor will be synthesized by the compiler, which runs the provided instance variable initializations. Similarly, if I did write a constructor but did not initialize all of the instance variables, each uninitialized instance variable would be initialized by its provided initializer. While this is mainly programmer convenience (one need only write the common initialization for an instance variable once, rather than once per non-delegating constructor), it may also have an impact on the overall initialization story (see below).

One- or Two-Phase Initialization?

Swift's current model it attempts to ensure that instance variables are initialized before they are used via definite initialization analysis. However, we haven't yet specified whether dynamic dispatch during initialization sees the object as being of the currently-executing constructor's type (as in C++/Java/C#) or of the final subclass's type (as in Objective-C).

I propose that we follow the C++/Java/C# precedent, which allows us to ensure (within Swift code) that an instance variable is never accessed before it has been initialized, eliminating the safety concerns introduced by Objective-C's two-phase initialization. Practically speaking, this means that the vtable/isa pointer should be set to the constructor's type while the constructor executes.

This model complicates constructor inheritance considerably. A secondary initializer in Objective-C works by delegating to (eventually) a designated initializer, which is overridden by the subclass. Following the C++/Java/C# precedent breaks this pattern, because the overriding designated initializer will never be invoked.

Constructor Inheritance

Currently, constructors in Swift are not inherited. This is a limitation that Swift's constructor model shares with C++98/03, Java, and C#, and a regression from Objective-C. C++11 introduced the notion of inherited constructors. It is an opt-in feature, introduced into one's class with a using declaration such as:

```
using MySuperclass::MySuperclass;
```

C++11 inherited constructors are implemented by essentially copying the signatures of all of the superclass's constructors into the subclass, ignoring those for which the subclass already has a constructor with the same signature. This approach does not translate well to Swift, because there is no way to gather all of the constructors in either the superclass or the subclass due to the presence of class extensions.

One potential approach is to bring Objective-C's notion of designated and secondary initializers into Swift. A "designated" constructor is responsible for calling the superclass constructor and then initializing its own instance variables.

A "secondary" constructor can be written in the class definition or an extension. A secondary constructor must delegate to another constructor, which will find the constructor to delegate to based on the type of the eventual subclass of the object. Thus, the subclass's override of the designated constructor will initialize all of the instance variables before continuing execution of the secondary constructor.

For the above to be safe, we need to ensure that subclasses override all of the designated constructors of their superclass. Therefore, we require that designated constructors be written within the class definition [1]. Secondary constructors can be written in either the class definition or an extension.

In Objective-C, classes generally only have one or two designated initializers, so having to override them doesn't seem too onerous. If it begins to feel like boilerplate, we could perform the override automatically when all of the instance variables of the subclass themselves have initializers.

Note that the requirement that all designated constructors be initializable means that adding a new designated constructor to a class is both an ABI- and API-breaking change.

Syntactically, we could introduce some kind of attribute or keyword to distinguish designated constructors from secondary constructors. Straw men: `[designated]` for designated constructors, which is implied for constructors in the class definition, and `[inherited]` for secondary constructors, which is implied for constructors in class extensions.

[1] We could be slightly more lenient here, allowing designated constructors to be written in any class extension within the same module as the class definition.

Class Clusters and Assignment to Self

TBD.

Objective-C Interoperability

The proposed Swift model for constructors needs to interoperate well with Objective-C, both for Objective-C classes imported into Swift and for Swift classes imported into Objective-C.

Constructor <-> Initializer Mapping

Fundamental to the interoperability story is that Swift constructors serve the same role as Objective-C initializers, and therefore should be interoperable. An Objective-C object should be constructible with Swift construction syntax, e.g.,:

```
NSDate()
```

and one should be able to override an Objective-C initializer in a Swift subclass by writing a constructor, e.g.,:

```
class Task : NSObject {
  constructor () {
    super.constructor() // invokes -[NSObject init]

    // perform initialization
  }
}
```

On the Objective-C side, one should be able to create a `Task` object with `[[Task alloc] init]`, subclass `Task` to override `-init`, and so on.

Each Swift constructor has both its two Swift entry points (one that allocates the object, one that initializes it) and an Objective-C entry point for the initializer. Given a Swift constructor, the selector for the Objective-C entry point is formed by:

- For the first selector piece, prepending the string "init" to the capitalized name of the first parameter.
- For the remaining selector pieces, the names of the remaining parameters.

For example, given the Swift constructor:

```
constructor withTitle(aTitle : String) date(aDate : NSDate) {
  // ...
}
```

the Objective-C entry point will have the selector `initWithTitle:date:`. Missing parameter names will be holes in the selector (e.g., "init::::"). If the constructor either has zero parameters or if it has a single parameter with `Void` type (i.e., the empty tuple `()`), the selector will be a zero-argument selector with no trailing colon, e.g.,:

```
constructor toMemory(_ : ()) { /* ... */ }
```

maps to the selector `initToMemory`.

This mapping is reversible: given a selector in the "init" family, i.e., where the first word is "init", we split the selector into its various pieces at the colons:

- For the first piece, we remove the "init" and then lowercase the next character *unless* the second character is also uppercase. This becomes the name of the first parameter to the constructor. If this string is non-empty and the selector is a zero-argument selector (i.e., no trailing :), the first (only) parameter has `Void` type.
- For the remaining pieces, we use the selector piece as the name of the corresponding parameter to the constructor.

Note that this scheme intentionally ignores methods that don't have the leading `init` keyword, including (e.g.) methods starting with `_init` or methods with completely different names that have been tagged as being in the `init` family in Objective-C (via the `objc_method_family` attribute in Clang). We consider these cases to be rare enough that we don't want to pessimize the conventional `init` methods to accommodate them.

Designated Initializers

Designed initializers in Objective-C are currently identified by documentation and convention. To strengthen these conventions and make it possible to mechanically verify (in Swift) that all designated initializers have been overridden, we should introduce a Clang method attribute `objc_designated_initializer` that can be applied to the designated initializers in Objective-C. Clang can then be extended to perform similar checking to what we're describing for Swift: designated initializers delegate or chain to the superclass constructor, secondary constructors always delegate, and subclassing requires one to override the designated initializers. The impact can be softened somewhat using warnings or other heuristics, to be (separately) determined.

Nil and Re-assigned Self

An Objective-C initializer can return a self pointer that is different than the one it was called with. When this happens, it is either due to an error (in which case it will return nil) or because the object is being substituted for another object.

In both cases, we are left with a partially-constructed object that then needs to be destroyed, even though its instance variables may not yet have been initialized. This is also a problem for Objective-C, which makes returning anything other than the original "self" brittle.

In Swift, we will have a separate error-handling mechanism to report failures. A Swift constructor will not be allowed to return a value; rather, it should raise an error if an error occurs, and that error will be propagated however we eventually decide to implement error propagation.

Object substitution is the more complicated feature. I propose that we do not initially support object substitution within Swift constructors. However, for memory safety we do need to recognize when calling the superclass constructor or delegating to another constructor has replaced the object (which can happen in Objective-C code). Aside from the need to destroy the original object, the instance variables of the new object will already have been initialized, so our "initialization" of those instance variables is actually assignment. The code generation for constructors will need to account for this.

Alternatives

This proposal is complicated, in part because it's trying to balance the safety goals of Swift against the convenience of Objective-C's two-phase initialization.

Separate Swift Constructors from Objective-C Initializers

Rather than try to adapt Swift's constructors to work with Objective-C's initializers, we could instead keep these features distinct. Swift constructors would maintain their current behavior (which ensures that instance variables are always initialized), and neither override nor introduce Objective-C initializers as entry points.

In this world, one would still have to implement `init` methods in Swift classes to override Objective-C initializers or make a Swift object constructible in Objective-C via the `alloc/init` pattern. Swift constructors would not be inherited, or would use some mechanism like C++11's inherited constructors. As we do today, Swift's Clang importer could introduce constructors into Objective-C classes that simply forward to the underlying initializers, so that we get Swift's object construction syntax. However, the need to write `init` methods when subclasses would feel like a kludge.

Embrace Two-Phase Initialization

We could switch Swift whole-heartedly over to two-phase initialization, eliminating constructors in favor of `init` methods. We would likely want to ensure that all instance variables get initialized before the `init` methods ever run, for safety reasons. This could be handled by (for example) requiring initializers on all instance variables, and running those initializers after allocation and before the `init` methods are called, the same way that instance variables with non-trivial default constructors are handled in Objective-C++. We would still likely need the notion of a designated initializer in Objective-C to make this safe in Swift, since we need to know which Objective-C initializers are guaranteed to initialize those instance variables not written in Swift.

This choice makes interoperability with Objective-C easier (since we're adopting Objective-C's model), but it makes safety either harder (e.g., we have to make all of our methods guard against uninitialized instance variables) or more onerous (requiring initializers on the declarations of all instance variables).