orphan: This proposal describes the work required to address rdar://problem/18216578.

Some terminology used below:

- **deallocating** refers to freeing the memory of an object without running any destructors.
- **releasing** refers to giving up a reference, which will result in running the destructor and deallocation of the object if this was the last reference.
- A **destructor** is a Swift-generated entry point which call the user-defined deinitializer, then releases all stored properties.
- A **deinitializer** is an optional user-defined entry point in a Swift class which handles any necessary cleanup beyond releasing stored properties.
- A **slice** of an object is the set of stored properties defined in one particular class forming the superclass chain of the instance.

Failable initializers

A **failable initializer** can return early with an error, without having initialized a new object. Examples can include initializers which validate input arguments, or attempt to acquire a limited resource.

There are two types of failable initializers:

- An initializer can be declared as having an optional return type, in which case it can signal failure by returning nil.
- An initializer can be declared as throwing, in which case it can signal failure by throwing an error.

Convenience initializers

Failing convenience initializers are the easy case, and are fully supported now. The failure can occur either before or after the self.init() delegation, and is handled as follows:

1. A failure prior to the self.init() delegation is handled by deallocating the completely-uninitialized self value.
2. A failure after the self.init() delegation is handled by releasing the fully-initialized self value.

Designated initializers

Failing designated initializers are more difficult, and are the subject of this proposal.

Similarly to convenience initializers, designated initializers can fail either before or after the super.init() delegation (or, for a root class initializer, the first location where all stored properties become initialized).

When failing after the super.init() delegation, we already have a fully-initialized self value, so releasing the self value is sufficient. The user-defined deinitializer, if any, is run in this case.

A failure prior to the super.init() delegation on the other hand will leave us with a partially-initialized self value that must be deallocated. We have to deinitialize any stored properties of self that we initialized, but we do not invoke the user-defined deinitializer method.

Description of the problem

To illustrate, say we are constructing an instance of a class C, and let superclasses(C) be the sequence of superclasses, starting from C and ending at a root class C_n:

```
superclasses(C) = {C, C_1, C_2, ..., C_n}
```

Suppose our failure occurs in the designated initializer for class C_k. At this point, the self value looks like this:

1. All stored properties in `{C, ..., C_(k-1)}` have been initialized.
2. Zero or more stored properties in `C_k` have been initialized.
3. The rest of the object `{C_(k+1), ..., C_n}` is completely uninitialized.

In order to fail out of the constructor without leaking memory, we have to destroy the initialized stored properties only without calling any Swift deinit methods, then deallocate the object itself.

There is a further complication once we take Objective-C interoperability into account. Objective-C classes can override -alloc, to get the object from a memory pool, for example. Also, they can override -retain and -release to implement their own reference counting. This means that if our class has @objc ancestry, we have to release it with -release even if it is partially initialized -- since this will result in Swift destructors being called, they have to know to skip the uninitialized parts of the object.

There is an issue we need to sort out, tracked by rdar://18720947. Basically, if we haven't done the `super.init()`, is it safe to call `-release`. The rest of this proposal assumes the answer is "yes".

Possible solutions

One approach is to think of the super.init() delegation as having a tri-state return value, instead of two-state:

1. First failure case -- object is fully initialized

2. Second failure case -- object is partially initialized
3. Success

This is problematic because now the ownership conventions in the initializer signature do not really describe the initializer's effect on reference counts; we now that this special return value for the second failure case, where the self value looks like it should have been consumed but it wasn't.

It is also difficult to encode this tri-state return for throwing initializers. One can imagine changing the try_apply and throw SIL instructions to support returning a pair (Error, AnyObject) instead of a single Error. But this would ripple changes throughout various SIL analyses, and require IRGen to encode the pair return value in an efficient way.

Proposed solution -- pure Swift case

A simpler approach seems to be to introduce a new partialDeinit entry point, referenced through a special kind of SILDeclRef. This entry point is dispatched through the vtable and invoked using a standard class_method sequence in SIL.

This entry point's job is to conditionally deinitialize stored properties of the self value, without invoking the user-defined deinitializer.

When a designated initializer for class C_k fails prior to performing the super.init() delegation, we emit the following code sequence:

1. First, de-initialize any stored properties this initializer may have initialized.
2. Second, invoke `partialDeinit(self, M)`, where M is the static metatype of `C_k`.

The partialDeinit entry point is implemented as follows:

1. If the static self type of the entry point is not equal to M, first delegate to the superclass's partialDeinit entry point, then deinitialize all stored properties in `C_k`.
2. If the static self type is equal to M, we have finished deinitializing the object, and we can now call a runtime function to deallocate it.

Note that we delegate to the superclass partialDeinit entry point before doing our own deinitialization, to ensure that stored properties are deinitialized in the reverse order in which they were initialized. This might not matter.

Note that if even if a class does not have any failing initializers of its own, it might delegate to a failing initializer in its superclass, using `super.init!` or `try!`. It might be easiest to emit a partialDeinit entry point for all classes, except those without any stored properties.

Proposed solution -- Objective-C case

As noted above, if the class has `@objc` ancestry, the interoperability story becomes more complicated. In order to undo any custom logic implemented in an Objective-C override of `-alloc` or `-retain`, we have to free the partially-initialized object using `-release`.

To ensure we don't double-free any Swift stored properties, we will add a new hidden stored property to each class that directly defines failing initializers. The bit is set if this slice of the instance has been initialized.

Note that unlike partialDeinit, if a class does not have failing initializers, it does not need this bit, even if its initializer delegates to a failing initializer in a superclass.

If the bit is clear, the destructor will skip the slice and not call the user-defined `deinit` method, or delegate further up the chain. Note that since newly-allocated Objective-C objects are zeroed out, the initial state of this bit indicates the slice is not initialized.

The constructor will set the bit before delegating to `super.init()`.

If a destructor fails before delegating to `super.init()`, it will call the partialDeinit entry point as before, but then, release the instance instead of deallocating it.

A possible optimization would be not generate the bit if all stored properties are POD, or retainable pointers. In the latter case, all zero bits is a valid representation (all the swift_retain/release entry points in the runtime check for null pointers, at least for now). However, we do not have to do this optimization right away.

Implementation

The bulk of this feature would be driven from DI. Right now, DI only implements failing designated initializers in their full generality for structs -- we already have logic for tracking which stored properties have been initialized, but the rest of the support for the partialDeinit entry point, as well as the Objective-C concerns needs to be fleshed out.