ARC Optimization for Swift

TODO: This is an evolving document on ARC optimization in the Swift compiler. Please extend it.

Contents

Terms

Some terms that are used often times in this document that must be defined. These may have more general definitions else where, but we define them with enough information for our purposes here:

1. Reference type: This is referring to a retainable pointer, not an aggregate that can contain a reference counted value.
2. A trivial type: A type for which a `retain_value` on a value of this type is a no-op.

Reference Counting Instructions

- `strong_retain`

- `strong_release`
- `strong_retain_unowned`
- `unowned_retain`
- `unowned_release`
- `load_weak`
- `store_weak`
- `fix_lifetime`
- `mark_dependence`
- `is_unique`
- `copy_block`

Memory Behavior of ARC Operations

At SIL level, reference counting and reference checking instructions are attributed with MayHaveSideEffects to prevent arbitrary passes from reordering them.

At IR level, retains are marked NoModRef with respect to load and store instructions so they don't pessimize memory dependence. (Note the Retains are still considered to write to memory with respect to other calls because getModRefBehavior is not overridden.) Releases cannot be marked NoModRef because they can have arbitrary side effects. `is_unique` calls cannot be marked `NoModRef` because they cannot be reordered with other operations that may modify the reference count.

TODO: Marking runtime calls with `NoModRef` in LLVM is misleading (they write memory), inconsistent (`getModRefBehavior` returns `Unknown`), and fragile (e.g. if we inline ARC operations at IR level). To be robust and allow stronger optimization, TBAA tags should be used to indicate functions that only access object metadata. This would also enable more LLVM level optimization in the presence of `is_unique` checks which currently appear to arbitrarily write memory.

ARC and Copying

TODO: Talk about how "ARC" and copying fit together. This means going into how retaining/releasing is really "copying"/"destroying" a pointer reference where the value that is pointed to does not change means you don't have to change the bits.

Talk about how this fits into `@owned` and `@guaranteed` parameters.

RC Identity

A core ARC concept in Swift optimization is the concept of `Reference Count Identity` (RC Identity) and RC Identity preserving instructions. In this section, we:

1. Define concepts related to RC identity.

2. Contrast RC identity analysis with alias analysis.
3. Discuss instructions/properties that cause certain instructions which "seem" to be RC identical to not be so.

Definitions

Let `I` be a SIL instruction with n operands and m results. We say that `I` is a (i, j) RC Identity preserving instruction if performing a `retain_value` on the ith SSA argument immediately before `I` is executed is equivalent to performing a `retain_value` on the jth SSA result of `I` immediately following the execution of `I`. For example in the following, if:

```
retain_value %x
%y = unary_instruction %x
```

is equivalent to:

```
%y = unary_instruction %x
retain_value %y
```

then we say that unary_instruction is a (0,0) RC Identity preserving instruction. In a case of a unary instruction, we omit (0,0) and just say that the instruction is RC Identity preserving.

TODO: This section defines RC identity only for loadable types. We also need to define it for instructions on addresses and instructions that mix addresses and values. It should be pretty straight forward to do this.

Given two SSA values `%a`, `%b`, we define `%a` as immediately RC identical to `%b` (or `%a ~rci %b`) if there exists an instruction `I` such that:

- `%a` is the jth result of `I`.
- `%b` is the ith argument of `I`.
- `I` is (i, j) RC identity preserving.

Due to the nature of SSA form, we can not even speak of symmetry or reflexivity. But we do get transitivity! Easily if `%b ~rci %a` and `%c ~rci %b`, we must by these two assumptions be able to do the following:

```
retain_value %a
%b = unary_instruction %a
%c = unary_instruction %b
```

which by our assumption means that we can perform the following code motion:

```
%b = unary_instruction %a
%c = unary_instruction %b
retain_value %c
```

our desired result. But we would really like for this operation to be reflexive and symmetric. To get around this issue, we define the equivalent relation RC identity as follows: We say that `%a ~rc %b` if:

1. `%a == %b`
2. `%a ~rci %b` or `%b ~rci %a`.
3. There exists a finite sequence of `n` SSA values `{%a[i]}` such that:
    - `%a ~rci %a[0]`
    - `%a[i] ~rci %a[i+1]` for all `i < n`.
    - `%a[n] ~rci %b`.

These equivalence classes consisting of chains of RC identical values are computed via the SILAnalysis called `RC Identity Analysis`. By performing ARC optimization on RC Identical operations, our optimizations are able to operate on the level of granularity that we actually care about, ignoring superficial changes in SSA form that still yield manipulations of the same reference count.

NOTE: `RCIdentityAnalysis` is a flow insensitive analysis. Dataflow that needs to be flow sensitive must handle phi nodes in the dataflow itself.

Contrasts with Alias Analysis

A common question is what is the difference in between RC Identity analysis and alias analysis. While alias analysis is attempting to determine if two memory locations are the same, RC identity analysis is attempting to determine if reference counting operations on different values would result in the same reference count being read or written to.

Some interesting examples of where RC identity differs from alias analysis are:

- `struct` is an RC identity preserving operation if the `struct` literal only has one non-trivial operand. This means for instance that any struct with one reference counted field used as an owning pointer is RC Identical with its owning pointer (a useful property for `Array`s).
- An `enum` instruction is always RC Identical with the given tuple payload.
- A `tuple` instruction is an RC identity preserving operation if the `tuple` literal has one non-trivial operand.
- `init_class_existential` is an RC identity preserving operation since performing a retain_value on a class existential is equivalent to performing a retain_value on the class itself.

The corresponding value projection operations have analogous properties.

NOTE: An important consequence of RC Identity is that value types with only one RCIdentity are a simple case for ARC optimization to handle. The ARC optimizer relies on other optimizations like SROA, Function Signature Opts, and SimplifyCFG (for block arguments) to try and eliminate cases where value types have multiple reference counted subtypes. If one has a struct type with multiple reference counted sub fields, wrapping the struct in a COW data structure (for instance storing the struct in an array of one element) will reduce the reference count overhead.

What is `retain_value` and why is it important?

Notice in the section above how we defined RC identity using the SIL `retain_value` instruction. `retain_value` and `release_value` are the catch-all please retain or please release this value at the SIL level. The following table is a quick summary of what `retain_value` (`release_value`) does when applied to various types of objects:

Ownership	Type	Effect
Strong	Class	Increment strong ref count of class
Any	Struct/Tuple	retain_value each field
Any	Enum	switch on the enum and apply `retain_value` to the enum case's payload (if it exists)
Unowned	Class	Increment the unowned ref count of class

Notice: Aggregate value types like struct/tuple/enums's definitions are defined recursively via retain_value on payloads/fields. This is why operations like `struct_extract` do not always propagate RC identity.

Conversions

Conversions are a common operation that propagate RC identity. But not all conversions have these properties. In this section, we attempt to explain why this is true. The rule for conversions is that a conversion that preserves RC identity must have the following properties:

1. Both of its arguments must be non-trivial values with the same ownership semantics (i.e. unowned, strong, weak). This means that the following conversions do not propagate RC identity:

   - `address_to_pointer`
   - `pointer_to_address`
   - `unchecked_trivial_bitcast`
   - `ref_to_raw_pointer`
   - `raw_pointer_to_ref`
   - `ref_to_unowned`
   - `unowned_to_ref`
   - `ref_to_unmanaged`
   - `unmanaged_to_ref`

   The reason why we want the ownership semantics to be the same is that whenever there is a change in ownership semantics, we want the programmer to explicitly reason about the change in ownership semantics.

2. The instruction must not introduce type aliasing. This disqualifies such casts as:

   - `unchecked_addr_cast`

- `unchecked_bitwise_cast`

This means in sum that conversions that preserve types and preserve non-trivialness are the interesting instructions.

ARC and Enums

Enum types provide interesting challenges for ARC optimization. This is because if there exists one case where an enum is non-trivial, the aggregate type in all situations must be treated as if it is non-trivial. An important consideration here is that when performing ARC optimization on cases, one has to be very careful about ensuring that one only ignores reference count operations on values that are able to be proved to be that specific case.

TODO: This section needs to be filled out more.

Copy-On-Write Considerations

The copy-on-write capabilities of some data structures, such as Array and Set, are efficiently implemented via Builtin.isUnique calls which lower directly to is_unique instructions in SIL.

The is_unique instruction takes the address of a reference, and although it does not actually change the reference, the reference must appear mutable to the optimizer. This forces the optimizer to preserve a retain distinct from what's required to maintain lifetime for any of the reference's source-level copies, because the called function is allowed to replace the reference, thereby releasing the referent. Consider the following sequence of rules:

1. An operation taking the address of a variable is allowed to replace the reference held by that variable. The fact that is_unique will not actually replace it is opaque to the optimizer.
2. If the refcount is 1 when the reference is replaced, the referent is deallocated.
3. A different source-level variable pointing at the same referent must not be changed/invalidated by such a call.
4. If such a variable exists, the compiler must guarantee the refcount is > 1 going into the call.

With the is_unique instruction, the variable whose reference is being checked for uniqueness appears mutable at the level of an individual SIL instruction. After IRGen, is_unique instructions are expanded into runtime calls that no longer take the address of the variable. Consequently, LLVM-level ARC optimization must be more conservative. It must not remove retain/release pairs of this form:

```
retain X
retain X
_swift_isUniquelyReferenced(X)
release X
release X
```

To prevent removal of the apparently redundant inner retain/release pair, the LLVM ARC optimizer should model `_swift_isUniquelyReferenced` as a function that may release X, use X, and exit the program (the subsequent release instruction does not prove safety).

`is_unique` instruction

As explained above, the SIL-level is_unique instruction enforces the semantics of uniqueness checks in the presence of ARC optimization. The kind of reference count checking that is_unique performs depends on the argument type:

- Native object types are directly checked by reading the strong reference count: (Builtin.NativeObject, known native class reference)
- Objective-C object types require an additional check that the dynamic object type uses native Swift reference counting: (unknown class reference, class existential)
- Bridged object types allow the dynamic object type check to be bypassed based on the pointer encoding: (Builtin.BridgeObject)

Any of the above types may also be wrapped in an optional. If the static argument type is optional, then a null check is also performed.

Thus, is_unique only returns true for non-null, native Swift object references with a strong reference count of one.

`Builtin.isUnique`

`Builtin.isUnique` gives the standard library access to optimization safe uniqueness checking. Because the type of reference check is derived from the builtin argument's static type, the most efficient check is automatically generated. However, in some cases, the standard library can dynamically determine that it has a native reference even though the static type is a bridge or unknown object. Unsafe variants of the builtin are available to allow the additional pointer bit mask and dynamic class lookup to be bypassed in these cases:

- `isUnique_native: <T> (inout T[?]) -> Int1`

These builtins perform an implicit cast to NativeObject before checking uniqueness. There's no way at SIL level to cast the address of a reference, so we need to encapsulate this operation as part of the builtin.

Semantic Tags

ARC takes advantage of certain semantic tags. This section documents these semantics and their meanings.

`programtermination_point`

If this semantic tag is applied to a function, then we know that:

- The function does not touch any reference counted objects.
- After the function is executed, all reference counted objects are leaked (most likely in preparation for program termination).

This allows one, when performing ARC code motion, to ignore blocks that contain an apply to this function as long as the block does not have any other side effect having instructions.

Unreachable Code and Lifetimes

The general case of unreachable code in terms of lifetime balancing has further interesting properties. Namely, an unreachable and `noreturn` functions signify a scope that has been split. This means that objects that are alive in that scope's lifetime may never end. This means that:

1. While we can not ignore all such unreachable terminated blocks for ARC purposes for instance, if we sink a retain past a br into a non `programtermination_point` block, we must sink the retain into the block.
2. If we are able to infer that an object's lifetime scope would never end due to the unreachable/no-return function, then we do not need to end the lifetime of the object early. An example of a situation where this can happen is with closure specialization. In closure specialization, we clone a caller that takes in a closure and create a copy of the closure in the caller with the specific closure. This allows for the closure to be eliminated in the specialized function and other optimizations to come into play. Since the lifetime of the original closure extended past any assertions in the original function, we do not need to insert releases in such locations to maintain program behavior.

ARC Sequence Optimization

TODO: Fill this in.

ARC Loop Hoisting

Abstract

This section describes the ARCLoopHoisting algorithm that hoists retains and releases out of loops. This is a high level description that justifies the correction of the algorithm and describes its design. In the following discussion we talk about the algorithm conceptually and show its safety and considerations necessary for good performance.

NOTE: In the following when we refer to "hoisting", we are not just talking about upward code motion of retains, but also downward code motion of releases.

Loop Canonicalization

In the following we assume that all loops are canonicalized such that:

1. The loop has a pre-header.
2. The loop has one backedge.
3. All exiting edges have a unique exit block.

Motivation

Consider the following simple loop:

```
bb0:
  br bb1

bb1:
  retain %x                    (1)
  apply %f(%x)
  apply %f(%x)
  release %x                   (2)
  cond_br ..., bb1, bb2

bb2:
  return ...
```

When it is safe to hoist (1),(2) out of the loop? Imagine if we know the trip count of the loop is 3 and completely unroll the loop so the whole function is one basic block. In such a case, we know the function looks as follows:

```
bb0:
  # Loop Iteration 0
  retain %x
  apply %f(%x)
  apply %f(%x)
  release %x                   (4)

  # Loop Iteration 1
  retain %x                    (5)
  apply %f(%x)
  apply %f(%x)
  release %x                   (6)

  # Loop Iteration 2
  retain %x                    (7)
  apply %f(%x)
  apply %f(%x)
  release %x
```

```
  return ...
```

Notice how (3) can be paired with (4) and (5) can be paired with (6). Assume that we eliminate those. Then the function looks as follows:

```
bb0:
  # Loop Iteration 0
  retain %x
  apply %f(%x)
  apply %f(%x)

  # Loop Iteration 1
  apply %f(%x)
  apply %f(%x)

  # Loop Iteration 2
  apply %f(%x)
  apply %f(%x)
  release %x

  return ...
```

We can then re-roll the loop, yielding the following loop:

```
bb0:
  retain %x                    (8)
  br bb1

bb1:
  apply %f(%x)
  apply %f(%x)
  cond_br ..., bb1, bb2

bb2:
  release %x                   (9)
  return ...
```

Notice that this transformation is equivalent to just hoisting (1) and (2) out of the loop in the original example. This form of hoisting is what is termed "ARCLoopHoisting". What is key to notice is that even though we are performing "hoisting" we are actually pairing releases from one iteration with retains in the next iteration and then eliminating the pairs. This realization will guide our further analysis.

Correctness

In this simple loop case, the proof of correctness is very simple to see conceptually. But in a more general case, when is safe to perform this optimization? We must

consider three areas of concern:

1. Are the retains/releases upon the same reference count? This can be found conservatively by using `RCIdentityAnalysis`.
2. Can we move retains, releases in the unrolled case as we have specified? This is simple since it is always safe to move a retain earlier and a release later in the dynamic execution of a program. This can only extend the life of a variable which is a legal and generally profitable in terms of allowing for this optimization.
3. How do we pair all necessary retains/releases to ensure we do not unbalance retain/release counts in the loop? Consider a set of retains and a set of releases that we wish to hoist out of a loop. We can only hoist the retain, release sets out of the loop if all paths in the given loop region from the entrance to the backedge have exactly one retain or release from this set.
4. Any early exits that we must move a retain past or a release by must be compensated appropriately. This will be discussed in the next section.

Assuming that our optimization does all of these things, we should be able to hoist with safety.

Compensating Early Exits for Lost Dynamic Reference Counts

Let's say that we have the following loop canonicalized SIL:

```
bb0(%0 : $Builtin.NativeObject):
  br bb1

bb1:
  strong_retain %0 : $Builtin.NativeObject
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject
  cond_br ..., bb2, bb3

bb2:
  cond_br ..., bb1, bb4

bb3:
  br bb5

bb4:
  br bb5

bb6:
  return ...
```

Can we hoist the retain/release pair here? Let's assume the loop is 3 iterations and we completely unroll it. Then we have:

```
bb0:
  strong_retain %0 : $Builtin.NativeObject               (1)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject              (2)
  cond_br ..., bb1, bb4

bb1: // preds: bb0
  strong_retain %0 : $Builtin.NativeObject               (3)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject              (4)
  cond_br ..., bb2, bb4

bb2: // preds: bb1
  strong_retain %0 : $Builtin.NativeObject               (5)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject              (6)
  cond_br ..., bb3, bb4

bb3: // preds: bb2
  br bb5

bb4: // preds: bb0, bb1, bb2
  br bb5

bb5: // preds: bb3, bb4
  return ...
```

We want to be able to pair and eliminate (2)/(3) and (4)/(5). In order to do that, we need to move (2) from `bb0` into `bb1` and (4) from `bb1` into `bb2`. In order to do this, we need to move a release along all paths into `bb4` lest we lose dynamic releases along that path. We also sink (6) in order to not have an extra release along that path. This then give us:

```
bb0:
  strong_retain %0 : $Builtin.NativeObject               (1)

bb1:
  apply %f(%0)
  apply %f(%0)
  cond_br ..., bb2, bb3

bb2:
  cond_br ..., bb1, bb4
```

```
bb3:
  strong_release %0 : $Builtin.NativeObject              (6*)
  br bb5

bb4:
  strong_release %0 : $Builtin.NativeObject              (7*)
  br bb5

bb5: // preds: bb3, bb4
  return ...
```

An easy inductive proof follows.

What if we have the opposite problem, that of moving a retain past an early exit. Consider the following:

```
bb0(%0 : $Builtin.NativeObject):
  br bb1

bb1:
  cond_br ..., bb2, bb3

bb2:
  strong_retain %0 : $Builtin.NativeObject
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject
  cond_br ..., bb1, bb4

bb3:
  br bb5

bb4:
  br bb5

bb6:
  return ...
```

Let's unroll this loop:

```
bb0(%0 : $Builtin.NativeObject):
  br bb1

# Iteration 1
bb1: // preds: bb0
  cond_br ..., bb2, bb8

bb2: // preds: bb1
```

```
  strong_retain %0 : $Builtin.NativeObject              (1)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject             (2)
  br bb3

# Iteration 2
bb3: // preds: bb2
  cond_br ..., bb4, bb8

bb4: // preds: bb3
  strong_retain %0 : $Builtin.NativeObject              (3)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject             (4)
  br bb5

# Iteration 3
bb5: // preds: bb4
  cond_br ..., bb6, bb8

bb6: // preds: bb5
  strong_retain %0 : $Builtin.NativeObject              (5)
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject             (6)
  cond_br ..., bb7, bb8

bb7: // preds: bb6
  br bb9

bb8: // Preds: bb1, bb3, bb5, bb6
  br bb9

bb9:
  return ...
```

First we want to move the retain into the previous iteration. This means that we have to move a retain over the `cond_br` in `bb1`, `bb3`, `bb5`. If we were to do that then `bb8` would have an extra dynamic retain along that path. In order to fix that issue, we need to balance that release by putting a release in `bb8`. But we cannot move a release into `bb8` without considering the terminator of `bb6` since `bb6` is also a predecessor of `bb8`. Luckily, we have (6). Notice that `bb7` has one predecessor to `bb6` so we can safely move 1 release along that path as well. Thus we perform that code motion, yielding the following:

```
bb0(%0 : $Builtin.NativeObject):
```

```
  br bb1

# Iteration 1
bb1: // preds: bb0
  strong_retain %0 : $Builtin.NativeObject              (1)
  cond_br ..., bb2, bb8

bb2: // preds: bb1
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject             (2)
  br bb3

# Iteration 2
bb3: // preds: bb2
  strong_retain %0 : $Builtin.NativeObject              (3)
  cond_br ..., bb4, bb8

bb4: // preds: bb3
  apply %f(%0)
  apply %f(%0)
  strong_release %0 : $Builtin.NativeObject             (4)
  br bb5

# Iteration 3
bb5: // preds: bb4
  strong_retain %0 : $Builtin.NativeObject              (5)
  cond_br ..., bb6, bb8

bb6: // preds: bb5
  apply %f(%0)
  apply %f(%0)
  cond_br ..., bb7, bb8

bb7: // preds: bb6
  strong_release %0 : $Builtin.NativeObject             (7*)
  br bb9

bb8: // Preds: bb1, bb3, bb5, bb6
  strong_release %0 : $Builtin.NativeObject             (8*)
  br bb9

bb9:
  return ...
```

Then we move (1), (3), (4) into the single predecessor of their parent block and

eliminate (3), (5) through a pairing with (2), (4) respectively. This yields then:

```
bb0(%0 : $Builtin.NativeObject):
  strong_retain %0 : $Builtin.NativeObject               (1)
  br bb1

# Iteration 1
bb1: // preds: bb0
  cond_br ..., bb2, bb8

bb2: // preds: bb1
  apply %f(%0)
  apply %f(%0)
  br bb3

# Iteration 2
bb3: // preds: bb2
  cond_br ..., bb4, bb8

bb4: // preds: bb3
  apply %f(%0)
  apply %f(%0)
  br bb5

# Iteration 3
bb5: // preds: bb4
  cond_br ..., bb6, bb8

bb6: // preds: bb5
  apply %f(%0)
  apply %f(%0)
  cond_br ..., bb7, bb8

bb7: // preds: bb6
  strong_release %0 : $Builtin.NativeObject              (7*)
  br bb9

bb8: // Preds: bb1, bb3, bb5, bb6
  strong_release %0 : $Builtin.NativeObject              (8*)
  br bb9

bb9:
  return ...
```

Then we finish by rerolling the loop:

```
bb0(%0 : $Builtin.NativeObject):
```

```
  strong_retain %0 : $Builtin.NativeObject               (1)
  br bb1

# Iteration 1
bb1: // preds: bb0
  cond_br ..., bb2, bb8

bb2:
  apply %f(%0)
  apply %f(%0)
  cond_br bb1, bb7

bb7:
  strong_release %0 : $Builtin.NativeObject              (7*)
  br bb9

bb8: // Preds: bb1, bb3, bb5, bb6
  strong_release %0 : $Builtin.NativeObject              (8*)
  br bb9

bb9:
  return ...
```

Uniqueness Check Complications

A final concern that we must consider is if we introduce extra copy on write copies through our optimization. To see this, consider the following simple IR sequence:

```
bb0(%0 : $Builtin.NativeObject):
  // refcount(%0) == n
  is_unique %0 : $Builtin.NativeObject
  // refcount(%0) == n
  strong_retain %0 : $Builtin.NativeObject
  // refcount(%0) == n+1
```

If n is not 1, then trivially `is_unique` will return false. So assume that n is 1 for our purposes so no copy is occurring here. Thus we have:

```
bb0(%0 : $Builtin.NativeObject):
  // refcount(%0) == 1
  is_unique %0 : $Builtin.NativeObject
  // refcount(%0) == 1
  strong_retain %0 : $Builtin.NativeObject
  // refcount(%0) == 2
```

Now imagine that we move the `strong_retain` before the `is_unique`. Then we have:

```
bb0(%0 : $Builtin.NativeObject):
  // refcount(%0) == 1
  strong_retain %0 : $Builtin.NativeObject
  // refcount(%0) == 2
  is_unique %0 : $Builtin.NativeObject
```

Thus `is_unique` is guaranteed to return false introducing a copy that was not needed. We wish to avoid that if it is at all possible.

Deinit Model

The semantics around deinits in swift are a common area of confusion. This section is not attempting to state where the deinit model may be in the future, but is just documenting where things are today in the hopes of improving clarity.

The following characteristics of deinits are important to the optimizer:

1. deinits run on the same thread and are not asynchronous like Java finalizers.
2. deinits are not sequenced with regards to each other or code in normal control flow.
3. If the optimizer takes advantage of the lack of sequencing it must do so in a way that preserves memory safety.

Consider the following pseudo-Swift example:

```
class D {}
class D1 : D {}
class D2 : D {}

var GLOBAL_D : D = D1()

class C { deinit { GLOBAL_D = D2() } }

func main() {
  let c = C()
  let d = GLOBAL_D
  useC(c)
  useD(d)
}

main()
```

Assume that `useC` does not directly in any way touch an instance of class `D` except via the destructor.

Since memory operations in normal control flow are not sequenced with respect to deinits, there are two correct programs here that the optimizer can produce: the original and the one where `useC(c)` and `GLOBAL_D` are swapped, i.e.:

```
func main() {
  let c = C()
  useC(c)
  let d = GLOBAL_D
  useD(d)
}
```

In the first program, `d` would be an instance of class `D1`. In the second, it would be an instance of class `D2`. Notice how in both programs though, no deinitialized object is accessed. On the other hand, imagine if we had split main like so:

```
func main() {
  let c = C()
  let d = unsafe_unowned_load(GLOBAL_D)
  useC(c)
  let owned_d = retain(d)
  useD(owned_d)
}
```

In this case, we would be passing off to `useD` a deallocated instance of class `D1` which would be undefined behavior. An optimization that produced such code would be a miscompile.