Chapter 6. Building Our Own "Arc"

In "Reference Counting" in Chapter 1, we saw the `std::sync::Arc<T>` type that allows for shared ownership through reference counting. The `Arc::new` function creates a new allocation, just like `Box::new`. However, unlike a `Box`, a cloned `Arc` will share the original allocation, without creating a new one. The shared allocation will only be dropped once the `Arc` and all its clones are dropped.

The memory ordering considerations involved in an implementation of this type can get quite interesting. In this chapter, we'll put more of the theory to practice by implementing our own `Arc<T>`. We'll start with a basic version, then extend it to support *weak pointers* for cyclic structures, and finish the chapter with an optimized version that's nearly identical to the implementation in the standard library.

Basic Reference Counting

Our first version will use a single `AtomicUsize` to count the number of `Arc` objects that share an allocation. Let's start with a struct that holds this counter and the `T` object:

```
struct ArcData<T> {
    ref_count: AtomicUsize,
    data: T,
}
```

Note that this struct is not public. It's an internal implementation detail of our `Arc` implementation.

Next is the `Arc<T>` struct itself, which is effectively just a pointer to a (shared) `ArcData<T>` object.

It might be tempting to make it a wrapper for a `Box<ArcData<T>>`, using a standard `Box` to handle the allocation of the `ArcData<T>`. However, a `Box` represents exclusive ownership, not shared ownership. We can't use a reference either, because we're not just borrowing the data owned by something else, and its lifetime ("until the last clone of this `Arc` is dropped") is not directly representable with a Rust lifetime.

Instead, we'll have to resort to using a pointer and handle allocation and the concept of ownership manually. Instead of a `*mut T` or `*const T`, we'll use a `std::ptr::NonNull<T>`, which represents a pointer to `T` that is never null. That way, an `Option<Arc<T>>` will be the same size as an `Arc<T>`, using the null pointer representation for `None`.

```
use std::ptr::NonNull;

pub struct Arc<T> {
    ptr: NonNull<ArcData<T>>,
}
```

With a reference or a `Box`, the compiler automatically understands for which `T` it should make your struct `Send` or `Sync`. When using a raw pointer or `NonNull`, however, it'll conservatively assume it's never `Send` or `Sync` unless we explicitly tell it otherwise.

Sending an `Arc<T>` across threads results in a `T` object being shared, requiring `T` to be `Sync`. Similarly, sending an `Arc<T>` across threads could result in another thread dropping that `T`, effectively transferring it to the other thread, requiring `T` to be `Send`. In other words, `Arc<T>` should be `Send` if and only if `T` is both `Send` and `Sync`. The exact same holds for `Sync`, since a shared `&Arc<T>` can be cloned into a new `Arc<T>`.

```
unsafe impl<T: Send + Sync> Send for Arc<T> {}
unsafe impl<T: Send + Sync> Sync for Arc<T> {}
```

For `Arc<T>::new`, we'll have to create a new allocation with an `ArcData<T>` with a reference count of one. We'll use `Box::new` to create a new allocation, `Box::leak` to give up our exclusive ownership of this allocation, and `NonNull::from` to turn it into a pointer:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
            ptr: NonNull::from(Box::leak(Box::new(ArcData {
                ref_count: AtomicUsize::new(1),
                data,
            }))),
        }
    }

    …
}
```

We know the pointer will always point to a valid `ArcData<T>` as long as the `Arc` object exists. However, this is not something the compiler knows or checks for us, so accessing the `ArcData` through the pointer requires unsafe code. We'll add a private helper function to get from the `Arc` to the `ArcData`, since this is something we'll have to do several times:

```
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }
```

Using that, we can now implement the `Deref` trait to make our `Arc<T>` transparently behave like a reference to a `T`:

```
impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.data().data
    }
}
```

Note that we don't implement `DerefMut`. Since an `Arc<T>` represents shared ownership, we can't unconditionally provide a `&mut T`.

Next: the `Clone` implementation. The cloned `Arc` will use the same pointer, after incrementing the reference counter:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        // TODO: Handle overflows.
        self.data().ref_count.fetch_add(1, Relaxed);
        Arc {
            ptr: self.ptr,
        }
    }
}
```

We can use `Relaxed` memory ordering to increment the reference counter, since there are no operations on other variables that need to strictly happen before or after this atomic operation. We already had access to the contained `T` before this operation

(through the original `Arc`), and that remains unchanged afterwards (but now through at least two `Arc` objects).

An `Arc` would need to be cloned many times before the counter has any chance of overflowing, but running `std::mem::forget(arc.clone())` in a loop can make it happen. We can use any of the techniques discussed in "Example: ID Allocation" in Chapter 2 and "Example: ID Allocation Without Overflow" in Chapter 2 to handle this issue.

To keep things as efficient as possible in the normal (non-overflowing) case, we'll keep the original `fetch_add` and simply abort the whole process if we get uncomfortably close to overflowing:

```
if self.data().ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
    std::process::abort();
}
```

> *Aborting the process is not instant, leaving for some time during which another thread can also call `Arc::clone`, incrementing the reference counter further. Therefore, just checking for `usize::MAX - 1` would not suffice. However, using `usize::MAX / 2` as the limit works fine: assuming every thread takes at least a few bytes of space in memory, it's impossible for `usize::MAX / 2` threads to exist concurrently.*

Just like we increment the counter when cloning, we need to decrement it when dropping an `Arc`. The thread that sees the counter go from one to zero knows it dropped the last `Arc<T>`, and is responsible for dropping and deallocating the `ArcData<T>`.

We'll use `Box::from_raw` to reclaim exclusive ownership of the allocation, and then drop it right away using `drop()`:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        // TODO: Memory ordering.
        if self.data().ref_count.fetch_sub(1, …) == 1 {
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
    }
}
```

For this operation, we can't use `Relaxed` ordering, since we need to make sure that nothing is still accessing the data when we drop it. In other words, every single drop of one of the former `Arc` clones must have happened before the final drop. So, the final `fetch_sub` must establish a happens-before relationship with every previous `fetch_sub` operation, which we can do using release and acquire ordering: decrementing it from, for example, two to one effectively "releases" the data, while decrementing it from one to zero "acquires" ownership of it.

We could use `AcqRel` memory ordering to cover both cases, but only the final decrement to zero needs `Acquire`, while the others only need `Release`. For efficiency, we'll use only `Release` for the `fetch_sub` operation and a separate `Acquire` fence only when necessary:

```
        if self.data().ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
```

Testing It

To test that our `Arc` is behaving as intended, we can write a unit test that creates an `Arc` containing a special object that lets us know when it gets dropped:

```
#[test]
fn test() {
    static NUM_DROPS: AtomicUsize = AtomicUsize::new(0);

    struct DetectDrop;

    impl Drop for DetectDrop {
        fn drop(&mut self) {
            NUM_DROPS.fetch_add(1, Relaxed);
        }
```

```
    }

    // Create two Arcs sharing an object containing a string
    // and a DetectDrop, to detect when it's dropped.
    let x = Arc::new(("hello", DetectDrop));
    let y = x.clone();

    // Send x to another thread, and use it there.
    let t = std::thread::spawn(move || {
        assert_eq!(x.0, "hello");
    });

    // In parallel, y should still be usable here.
    assert_eq!(y.0, "hello");

    // Wait for the thread to finish.
    t.join().unwrap();

    // One Arc, x, should be dropped by now.
    // We still have y, so the object shouldn't have been dropped yet.
    assert_eq!(NUM_DROPS.load(Relaxed), 0);

    // Drop the remaining `Arc`.
    drop(y);

    // Now that `y` is dropped too,
    // the object should've been dropped.
    assert_eq!(NUM_DROPS.load(Relaxed), 1);
}
```

This compiles and runs fine, so it seems our `Arc` is behaving as intended! While this is encouraging, it doesn't prove that the implementation is fully correct. It's advisable to use a long stress test involving many threads to gain more confidence.

Miri

It can also be very useful to run tests using Miri. Miri is an experimental but very useful and powerful tool to check unsafe code for various forms of undefined behavior.

Miri is an interpreter for the Rust compiler's mid-level intermediate representation. This means that it runs your code not by compiling it to native processor instructions, but instead by interpreting it at a point when information like types and lifetimes are still available. Because of this, Miri runs programs significantly slower than when they are compiled and run normally, but is able to detect many mistakes that would result in undefined behavior.

It includes experimental support for detecting data races, which allows it to detect memory ordering problems.

For more details and a guide on how to use Miri, see its GitHub page.

Mutation

As mentioned before, we can't implement `DerefMut` for our `Arc`. We can't unconditionally promise exclusive access (`&mut T`) to the data, because it might be accessed through other `Arc` objects.

However, what we can do is to allow it conditionally. We can make a method that only gives out a `&mut T` if the reference counter is one, proving that there's no other `Arc` object that could be used to access the same data.

This function, which we'll call `get_mut`, will have to take a `&mut Self` to make sure nothing else can use this same `Arc` to access the `T`. Knowing that there's only one `Arc` would be meaningless if that one `Arc` can still be shared.

We'll need to use acquire memory ordering to make sure that threads that previously owned a clone of the `Arc` are no longer accessing the data. We need to establish a happens-before relationship with every single `drop` that led to the reference counter being one.

This only matters when the reference counter is actually one; if it's higher, we'll not provide a `&mut T`, and the memory ordering is irrelevant. So, we can use a relaxed load, followed by a conditional acquire fence, as follows:

```
pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
    if arc.data().ref_count.load(Relaxed) == 1 {
        fence(Acquire);
        // Safety: Nothing else can access the data, since
        // there's only one Arc, to which we have exclusive access.
        unsafe { Some(&mut arc.ptr.as_mut().data) }
    } else {
        None
    }
}
```

This function does not take a `self` argument, but takes a regular argument (named `arc`) instead. This means it can only be called as `Arc::get_mut(&mut a)`, and not as `a.get_mut()`. This is advisable for types that implement `Deref`, to avoid ambiguity with a similarly named method on the underlying `T`.

The returned mutable reference implicitly borrows the lifetime from the argument, meaning that nothing can use the original `Arc` as long as the returned `&mut T` is still around, allowing for safe mutation.

When the lifetime of the `&mut T` expires, the `Arc` can be used and shared with other threads again. One might wonder whether we need to worry about memory ordering for threads accessing the data afterwards. However, that's the responsibility of whatever mechanism is used for sharing the `Arc` (or a new clone of it) with another thread. (For example, a mutex, a channel, or spawning a new thread.)

Weak Pointers

Reference counting can be very useful when representing structures in memory consisting of multiple objects. For example, every node in a tree structure could contain an `Arc` to each of its child nodes. That way, when a node is dropped, its child nodes that are no longer in use are all (recursively) dropped as well.

This breaks down for *cyclic structures*, however. If a child node also contains an `Arc` to its parent node, neither will be dropped since there's always at least one `Arc` that still refers to it.

The standard library's `Arc` comes with a solution for that problem: `Weak<T>`. A `Weak<T>`, also called a *weak pointer*, behaves a bit like an `Arc<T>`, but does not prevent an object from getting dropped. A `T` can be shared between several `Arc<T>` and `Weak<T>` objects, but when all `Arc<T>` objects are gone, the `T` is dropped, regardless of whether there are any `Weak<T>` objects left.

This means that a `Weak<T>` can exist without a `T`, and thus cannot provide a `&T` unconditionally, like an `Arc<T>` can. However, to access the `T` given a `Weak<T>`, it can be *upgraded* to an `Arc<T>` through its `upgrade()` method. This method returns an `Option<Arc<T>>`, returning `None` if the `T` has already been dropped.

In an `Arc`-based structure, `Weak` can be used to break cycles. For example, child nodes in a tree structure could use `Weak` rather than `Arc` for their parent node. Then, dropping of a parent node is not prevented through the existence of its child nodes.

Let's implement this.

Just like before, when the number of `Arc` objects reaches zero, we can drop the contained `T` object. However, we can't drop and deallocate the `ArcData` yet, since there might still be weak pointers referencing it. Only once the last `Weak` pointer is also gone can we drop and deallocate the `ArcData`.

So, we'll use two counters: one for "the number of things that reference the `T`," and another for "the number of things that reference the `ArcData<T>`." In other words, the first counter is the same as before: it counts `Arc` objects, while the second counter counts both `Arc` and `Weak` objects.

We also need something that allows us to drop the contained object (`T`) while the `ArcData<T>` is still in use by the weak pointers. We'll use an `Option<T>` so we can use `None` for when the data is dropped, and wrap that in an `UnsafeCell` for *interior mutability* ("Interior Mutability" in Chapter 1), to allow that to happen when the `ArcData<T>` isn't exclusively owned:

```
struct ArcData<T> {
    /// Number of `Arc`s.
```

```
    data_ref_count: AtomicUsize,
    /// Number of `Arc`s and `Weak`s combined.
    alloc_ref_count: AtomicUsize,
    /// The data. `None` if there's only weak pointers left.
    data: UnsafeCell<Option<T>>,
}
```

If we think of a `Weak<T>` as an object responsible for keeping an `ArcData<T>` alive, it can make sense to implement `Arc<T>` as a struct containing a `Weak<T>`, since an `Arc<T>` needs to do the same, and more.

```
pub struct Arc<T> {
    weak: Weak<T>,
}

pub struct Weak<T> {
    ptr: NonNull<ArcData<T>>,
}

unsafe impl<T: Sync + Send> Send for Weak<T> {}
unsafe impl<T: Sync + Send> Sync for Weak<T> {}
```

The `new` function is mostly the same as before, except it now has two counters to initialize at once:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
```

```
                weak: Weak {
                    ptr: NonNull::from(Box::leak(Box::new(ArcData {
                        alloc_ref_count: AtomicUsize::new(1),
                        data_ref_count: AtomicUsize::new(1),
                        data: UnsafeCell::new(Some(data)),
                    }))),
                },
            }
        }

        …
    }
```

Just like before, we assume that the `ptr` field always points at a valid `ArcData<T>`. This time, we'll encode that assumption as a private `data()` helper method on `Weak<T>`:

```
impl<T> Weak<T> {
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }

    …
}
```

In the `Deref` implementation for `Arc<T>`, we now have to use `UnsafeCell::get()` to get a pointer to the contents of the cell, and use unsafe code to promise it can safely be shared at this point. We also need `as_ref().unwrap()` to get a reference into the

`Option<T>`. We don't have to worry about this panicking, since the `Option` will only be `None` when there are no `Arc` objects left.

```
impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        let ptr = self.weak.data().data.get();
        // Safety: Since there's an Arc to the data,
        // the data exists and may be shared.
        unsafe { (*ptr).as_ref().unwrap() }
    }
}
```

The `Clone` implementation for `Weak<T>` is quite straightforward; it's pretty much identical to our previous `Clone` implementation for `Arc<T>`:

```
impl<T> Clone for Weak<T> {
    fn clone(&self) -> Self {
        if self.data().alloc_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Weak { ptr: self.ptr }
    }
}
```

In the `Clone` implementation for our new `Arc<T>`, we need to increment both counters. We'll simply use `self.weak.clone()` to reuse the code above for the first counter, so we only have to manually increment the second counter:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        let weak = self.weak.clone();
        if weak.data().data_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Arc { weak }
    }
}
```

Dropping a `Weak` should decrement its counter and drop and deallocate the `ArcData` when the counter goes from one to zero. This is identical to what the `Drop` implementation of our previous `Arc` did.

```
impl<T> Drop for Weak<T> {
    fn drop(&mut self) {
        if self.data().alloc_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
```

```
    }
}
```

Dropping an `Arc` should decrement both counters. Note that one of these is already automatically taken care of, since every `Arc` contains a `Weak`, such that dropping an `Arc` will also result in dropping a `Weak`. We only have to take care of the other counter:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        if self.weak.data().data_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            let ptr = self.weak.data().data.get();
            // Safety: The data reference counter is zero,
            // so nothing will access it.
            unsafe {
                (*ptr) = None;
            }
        }
    }
}
```

> *Dropping an object in Rust will first run its `Drop::drop` function (if it implements `Drop`), and then drop all of its fields, one by one, recursively.*

The check in the `get_mut` method remains mostly unchanged, except it now needs to take weak pointers into account. It might seem like it could ignore weak pointers when checking for exclusivity, but a `Weak<T>` can be upgraded to an `Arc<T>` at any time. So,

`get_mut` will have to check that there are no other `Arc<T>` or `Weak<T>` pointers before it can give out a `&mut T`:

```
impl<T> Arc<T> {
    …

    pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
        if arc.weak.data().alloc_ref_count.load(Relaxed) == 1 {
            fence(Acquire);
            // Safety: Nothing else can access the data, since
            // there's only one Arc, to which we have exclusive access,
            // and no Weak pointers.
            let arcdata = unsafe { arc.weak.ptr.as_mut() };
            let option = arcdata.data.get_mut();
            // We know the data is still available since we
            // have an Arc to it, so this won't panic.
            let data = option.as_mut().unwrap();
            Some(data)
        } else {
            None
        }
    }

    …
}
```

Next up: upgrading a weak pointer. Upgrading a `Weak` to an `Arc` is only possible when the data still exists. If there are only weak pointers left, there's no data left that can be shared through an `Arc`. So, we'll have to increase the `Arc` counter, but can only do so if

it isn't already zero. We'll use a compare-and-exchange loop ("Compare-and-Exchange Operations" in Chapter 2) to do this.

Just like before, relaxed memory ordering is fine for incrementing a reference counter. There are no operations on other variables that need to strictly happen before or after this atomic operation.

```
impl<T> Weak<T> {
    …

    pub fn upgrade(&self) -> Option<Arc<T>> {
        let mut n = self.data().data_ref_count.load(Relaxed);
        loop {
            if n == 0 {
                return None;
            }
            assert!(n < usize::MAX);
            if let Err(e) =
                self.data()
                    .data_ref_count
                    .compare_exchange_weak(n, n + 1, Relaxed, Relaxed)
            {
                n = e;
                continue;
            }
            return Some(Arc { weak: self.clone() });
        }
    }
}
```

Note how this time we can check for `n < usize::MAX`, since that assertion would panic before we modify `data_ref_count`.

The opposite, getting a `Weak<T>` from an `Arc<T>`, is much simpler:

```
impl<T> Arc<T> {
    …

    pub fn downgrade(arc: &Self) -> Weak<T> {
        arc.weak.clone()
    }
}
```

Testing It

To quickly test our creation, we'll modify our previous unit test to use weak pointers and verify that they can be upgraded when expected:

```
#[test]
fn test() {
    static NUM_DROPS: AtomicUsize = AtomicUsize::new(0);

    struct DetectDrop;

    impl Drop for DetectDrop {
        fn drop(&mut self) {
            NUM_DROPS.fetch_add(1, Relaxed);
        }
```

```
    }

    // Create an Arc with two weak pointers.
    let x = Arc::new(("hello", DetectDrop));
    let y = Arc::downgrade(&x);
    let z = Arc::downgrade(&x);

    let t = std::thread::spawn(move || {
        // Weak pointer should be upgradable at this point.
        let y = y.upgrade().unwrap();
        assert_eq!(y.0, "hello");
    });
    assert_eq!(x.0, "hello");
    t.join().unwrap();

    // The data shouldn't be dropped yet,
    // and the weak pointer should be upgradable.
    assert_eq!(NUM_DROPS.load(Relaxed), 0);
    assert!(z.upgrade().is_some());

    drop(x);

    // Now, the data should be dropped, and the
    // weak pointer should no longer be upgradable.
    assert_eq!(NUM_DROPS.load(Relaxed), 1);
    assert!(z.upgrade().is_none());
}
```

This also compiles and runs without problems, which leaves us with a very usable handmade `Arc` implementation.

Optimizing

While weak pointers can be useful, the `Arc` type is often used without any weak pointers. A downside of our last implementation is that cloning and dropping an `Arc` now both take two atomic operations each, as they have to increment or decrement both counters. This makes all `Arc` users "pay" for the cost of weak pointers, even when they are not using them.

It might seem like the solution is to count `Arc<T>` and `Weak<T>` pointers separately, but then we wouldn't be able to atomically check that both counters are zero. To understand how that's a problem, imagine we have a thread executing the following annoying function:

```
fn annoying(mut arc: Arc<Something>) {
    loop {
        let weak = Arc::downgrade(&arc);
        drop(arc);
        println!("I have no Arc!");  1
        arc = weak.upgrade().unwrap();
        drop(weak);
        println!("I have no Weak!");  2
    }
}
```

This thread continuously downgrades and upgrades an `Arc`, such that it repeatedly cycles through moments where it holds no `Arc` (`1`), and moments where it holds no `Weak` (`2`). If we check both counters to see if there are any threads still using the allocation, this thread might be able to hide its existence if we are unlucky and check the `Arc` counter during its first print statement (`1`), but check the `Weak` counter during its second print statement (`2`).

In our last implementation, we solved this by counting every `Arc` also as a `Weak`. A more subtle way of solving this is to count all `Arc` pointers combined as one single `Weak` pointer. That way, the weak pointer counter (`alloc_ref_count`) never reaches zero as long as there is still at least one `Arc` object around, just like in our last implementation, but cloning an `Arc` doesn't need to touch that counter at all. Only dropping the very last `Arc` will decrement the weak pointer counter too.

Let's try that.

This time, we can't just implement `Arc<T>` as a wrapper around `Weak<T>`, so both will wrap a non-null pointer to the allocation:

```
pub struct Arc<T> {
    ptr: NonNull<ArcData<T>>,
}

unsafe impl<T: Sync + Send> Send for Arc<T> {}
unsafe impl<T: Sync + Send> Sync for Arc<T> {}
```

```
pub struct Weak<T> {
    ptr: NonNull<ArcData<T>>,
}

unsafe impl<T: Sync + Send> Send for Weak<T> {}
unsafe impl<T: Sync + Send> Sync for Weak<T> {}
```

Since we're optimizing our implementation, we might as well make `ArcData<T>` slightly smaller by using a `std::mem::ManuallyDrop<T>` instead of an `Option<T>`. We used an `Option<T>` to be able to replace a `Some(T)` by `None` when dropping the data, but we don't actually need a separate `None` state to tell us the data is gone, since the existence or absence of `Arc<T>` already tells us that. A `ManuallyDrop<T>` takes the exact same amount of space as a `T`, but allows us to manually drop it at any point by using an unsafe call to `ManuallyDrop::drop()`:

```
use std::mem::ManuallyDrop;

struct ArcData<T> {
    /// Number of `Arc`s.
    data_ref_count: AtomicUsize,
    /// Number of `Weak`s, plus one if there are any `Arc`s.
    alloc_ref_count: AtomicUsize,
    /// The data. Dropped if there are only weak pointers left.
    data: UnsafeCell<ManuallyDrop<T>>,
}
```

The `Arc::new` function remains almost unchanged, initializing both counters at one like before, but now using `ManuallyDrop::new()` instead of `Some()`:

```
impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T> {
        Arc {
            ptr: NonNull::from(Box::leak(Box::new(ArcData {
                alloc_ref_count: AtomicUsize::new(1),
                data_ref_count: AtomicUsize::new(1),
                data: UnsafeCell::new(ManuallyDrop::new(data)),
            }))),
        }
    }

    …
}
```

The `Deref` implementation can no longer make use of the private `data` method on the `Weak` type, so we'll add the same private helper function on `Arc<T>`:

```
impl<T> Arc<T> {
    …

    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }

    …
}
```

```
impl<T> Deref for Arc<T> {
    type Target = T;

    fn deref(&self) -> &T {
        // Safety: Since there's an Arc to the data,
        // the data exists and may be shared.
        unsafe { &*self.data().data.get() }
    }
}
```

The `Clone` and `Drop` implementations for `Weak<T>` remain exactly the same as for our last implementation. Here they are for completeness, including the private `Weak::data` helper function:

```
impl<T> Weak<T> {
    fn data(&self) -> &ArcData<T> {
        unsafe { self.ptr.as_ref() }
    }

    …
}

impl<T> Clone for Weak<T> {
    fn clone(&self) -> Self {
        if self.data().alloc_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Weak { ptr: self.ptr }
    }
```

```
}

impl<T> Drop for Weak<T> {
    fn drop(&mut self) {
        if self.data().alloc_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            unsafe {
                drop(Box::from_raw(self.ptr.as_ptr()));
            }
        }
    }
}
```

And now we get to the part that this new optimized implementation was all about—cloning an `Arc<T>` now needs to touch only one counter:

```
impl<T> Clone for Arc<T> {
    fn clone(&self) -> Self {
        if self.data().data_ref_count.fetch_add(1, Relaxed) > usize::MAX / 2 {
            std::process::abort();
        }
        Arc { ptr: self.ptr }
    }
}
```

Similarly, dropping an `Arc<T>` now needs to decrement only one counter, except for the last drop that sees that counter go from one to zero. In that case, the weak pointer counter also needs to be decremented, such that it can reach zero once there are no

weak pointers left. We do this by simply creating a `Weak<T>` out of thin air and immediately dropping it:

```
impl<T> Drop for Arc<T> {
    fn drop(&mut self) {
        if self.data().data_ref_count.fetch_sub(1, Release) == 1 {
            fence(Acquire);
            // Safety: The data reference counter is zero,
            // so nothing will access the data anymore.
            unsafe {
                ManuallyDrop::drop(&mut *self.data().data.get());
            }
            // Now that there's no `Arc<T>`s left,
            // drop the implicit weak pointer that represented all `Arc<T>`s.
            drop(Weak { ptr: self.ptr });
        }
    }
}
```

The `upgrade` method on `Weak<T>` remains mostly the same, except it no longer clones a weak pointer, since it doesn't need to increment the weak pointer counter anymore. Upgrading only succeeds if there is already at least one `Arc<T>` to the allocation, which means that `Arc`s are already accounted for in the weak pointer counter.

```
impl<T> Weak<T> {
    …

    pub fn upgrade(&self) -> Option<Arc<T>> {
```

```
            let mut n = self.data().data_ref_count.load(Relaxed);
            loop {
                if n == 0 {
                    return None;
                }
                assert!(n < usize::MAX);
                if let Err(e) =
                    self.data()
                        .data_ref_count
                        .compare_exchange_weak(n, n + 1, Relaxed, Relaxed)
                {
                    n = e;
                    continue;
                }
                return Some(Arc { ptr: self.ptr });
            }
        }
    }
```

So far the differences between this and our previous implementation are very minimal. Where things get tricky, though, is with the last two methods we still need to implement: `downgrade` and `get_mut`.

Unlike before, the `get_mut` method now needs to check if both counters are set to one to be able to determine whether there's only one `Arc<T>` and no `Weak<T>` left, since a weak pointer counter of one can now represent multiple `Arc<T>` pointers. Reading the counters are two separate operations that happen at (slightly) different times, so we

have to be very careful to not miss any concurrent downgrades, such as in the example case we saw at the start of "Optimizing".

If we first check that `data_ref_count` is one, then we could miss a subsequent `upgrade()` before we check the other counter. But, if we first check that `alloc_ref_count` is one, then we could miss a subsequent `downgrade()` before we check the other counter.

A way out of this dilemma is to briefly block the `downgrade()` operation by "locking" the weak pointer counter. To do that, we don't need anything like a mutex. We can use a special value, like `usize::MAX`, to represent a special "locked" state of the weak pointer counter. It'll only be locked very briefly, only to load the other counter, so the `downgrade` method could just spin until it's unlocked, in the unlikely situation it runs at the exact same moment as `get_mut`.

So, in `get_mut` we'll first have to check if `alloc_ref_count` is one and at the same time replace it by `usize::MAX`, if it was indeed one. That's a job for `compare_exchange`.

Then we'll have to check if the other counter is also one, after which we can immediately unlock the weak pointer counter. If the second counter is also one, we know we have exclusive access to the allocation and the data, such that we can return a `&mut T`.

```
pub fn get_mut(arc: &mut Self) -> Option<&mut T> {
    // Acquire matches Weak::drop's Release decrement, to make sure any
```

```
        // upgraded pointers are visible in the next data_ref_count.load.
        if arc.data().alloc_ref_count.compare_exchange(
            1, usize::MAX, Acquire, Relaxed
        ).is_err() {
            return None;
        }
        let is_unique = arc.data().data_ref_count.load(Relaxed) == 1;
        // Release matches Acquire increment in `downgrade`, to make sure any
        // changes to the data_ref_count that come after `downgrade` don't
        // change the is_unique result above.
        arc.data().alloc_ref_count.store(1, Release);
        if !is_unique {
            return None;
        }
        // Acquire to match Arc::drop's Release decrement, to make sure nothing
        // else is accessing the data.
        fence(Acquire);
        unsafe { Some(&mut *arc.data().data.get()) }
    }
```

As you might have expected, the locking operation (the `compare_exchange`) will have to use `Acquire` memory ordering, and the unlocking operation (the `store`) will have to use `Release` memory ordering.

If we had used `Relaxed` for the `compare_exchange` instead, it would have been possible for the subsequent `load` from `data_ref_count` to not see the new value of a freshly upgraded `Weak` pointer, even though the `compare_exchange` had already confirmed that every `Weak` pointer had been dropped.

If we had used `Relaxed` for the `store`, it would have been possible for the preceding `load` to observe the result of a future `Arc::drop` for an `Arc` that can still be downgraded.

The acquire fence is the same as before: it synchronizes with the release-decrement operation in `Arc::Drop` to make sure every access through former `Arc` clones has happened before the new exclusive access.

The last piece of the puzzle is the `downgrade` method, which will have to check for the special `usize::MAX` value to see if the weak pointer counter is locked, and spin until it is unlocked. Just like in the `upgrade` implementation, we'll use a compare-and-exchange loop to check for the special value and overflow before incrementing the counter:

```
pub fn downgrade(arc: &Self) -> Weak<T> {
    let mut n = arc.data().alloc_ref_count.load(Relaxed);
    loop {
        if n == usize::MAX {
            std::hint::spin_loop();
            n = arc.data().alloc_ref_count.load(Relaxed);
            continue;
        }
        assert!(n < usize::MAX - 1);
        // Acquire synchronises with get_mut's release-store.
        if let Err(e) =
            arc.data()
                .alloc_ref_count
```

```
                    .compare_exchange_weak(n, n + 1, Acquire, Relaxed)
            {
                n = e;
                continue;
            }
            return Weak { ptr: arc.ptr };
        }
    }
```

We use acquire memory ordering for `compare_exchange_weak`, which synchronizes with the release-store in the `get_mut` function. Otherwise, it would be possible for the effect of a subsequent `Arc::drop` to be visible to a thread running `get_mut` before it unlocks the counter.

In other words, the acquire compare-and-exchange operation here effectively "locks" `get_mut`, preventing it from succeeding. It can be "unlocked" again by a later `Weak::drop` that decrements the counter back to one, using release memory ordering.

> *The optimized implementation of `Arc<T>` and `Weak<T>` that we just made is nearly identical to the one included in the Rust standard library.*

If we run the exact same test as before ("Testing It"), we see that this optimized implementation also compiles and passes our tests.

> *If you feel that getting the memory ordering decisions right for this optimized implementation was difficult, don't worry. Many concurrent data structures are*

> *simpler to implement correctly than this one. This `Arc` implementation is included in this chapter specifically because of its tricky subtleties around memory ordering.*

~

Summary

- `Arc<T>` provides shared ownership of a reference-counted allocation.
- By checking if the reference counter is exactly one, an `Arc<T>` can conditionally provide exclusive access (`&mut T`).
- Incrementing the atomic reference counter can be done using a relaxed operation, but the final decrement must synchronize with all previous decrements.
- A *weak pointer* (`Weak<T>`) can be used to avoid cycles.
- The `NonNull<T>` type represents a pointer to `T` that is never null.
- The `ManuallyDrop<T>` type can be used to manually decide, using `unsafe` code, when to drop a `T`.

- As soon as more than one atomic variable is involved, things get more complicated.
- Implementing an ad hoc (spin) lock can sometimes be a valid strategy for operating on multiple atomic variables at once.

Next: Chapter 7. Understanding the Processor