Rust-101, Part 12: Rc, Interior Mutability, Cell, RefCell

Our generic callback mechanism is already working quite nicely. However, there's one point we may want to fix: `Callbacks` does not implement `Clone`. The problem is that closures (or rather, their environment) can never be cloned. (There's not even an automatic derivation happening for the cases where it would be possible.) This restriction propagates up to `Callbacks` itself. What could we do about this?

Rc

The solution is to find some way of cloning `Callbacks` without cloning the environments. This can be achieved with `Rc<T>`, a *reference-counted* pointer. This is another example of a smart pointer. You can `clone` an `Rc` as often as you want, that doesn't affect the data it contains. It only creates more references to the same data. Once all the references are gone, the data is deleted.

Wait a moment, you may say here. Multiple references to the same data? That's aliasing! Indeed: Once data is stored in an `Rc`, it is read-only and you can only ever get a shared reference to the data again.

Because of this read-only restriction, we cannot use `FnMut` here: We'd be unable to call the function with a mutable reference to it's environment! So we have to go with `Fn`. We wrap that in an `Rc`, and then Rust happily derives `Clone` for us.

```
#[derive(Clone)]
struct Callbacks {
    callbacks: Vec<Rc<Fn(i32)>>,
}

impl Callbacks {
    pub fn new() -> Self {
        Callbacks { callbacks: Vec::new() }
    }

    pub fn register<F: Fn(i32)+'static>(&mut self, callback: F) {
        self.callbacks.push(Rc::new(callback));
    }

    pub fn call(&self, val: i32) {
        for callback in self.callbacks.iter() {
            callback(val);
        }
    }
}
fn demo(c: &mut Callbacks) {
    c.register(|val| println!("Callback 1: {}", val));
    c.call(0); c.clone().call(1);
}

pub fn main() {
    let mut c = Callbacks::new();
    demo(&mut c);
}
```

Registration works just like last time, except that we are creating an `Rc` now.

We only need a shared iterator here. Since `Rc` is a smart pointer, we can directly call the callback.

Time for a demo!

Interior Mutability

Of course, the counting example from last time does not work anymore: It needs to mutate the environment, which a `Fn` cannot do. The strict borrowing Rules of Rust are getting into our way. However, when it comes to mutating a mere number (`usize`), there's not really any chance of problems coming up. Everybody can read and write that variable just as they want. So it would be rather sad if we were not able to write this program. Lucky enough, Rust's standard library provides a solution in the form of `Cell<T>`. This type represents a memory cell of some type `T`, providing the two basic operations `get` and `set`. `get` returns a *copy* of the content of the cell, so all this works only if `T` is `Copy`. `set`, which overrides the content, only needs a *shared reference* to the cell. The phenomenon of a type that permits mutation through shared references (i.e., mutation despite the possiblity of aliasing) is called *interior mutability*. You can think of `set` changing only the *contents* of the cell, not its *identity*. In contrast, the kind of mutation we saw so far was about replacing one piece of data by something else of the same type. This is called *inherited mutability*.
Notice that it is impossible to *borrow* the contents of the cell, and that is actually the key to why this is safe.

So, let us put our counter in a `Cell`, and replicate the example from the previous part.

Again, we have to move ownership of the `count` into the environment closure.

In here, all we have is a shared reference of our environment. But that's good enough for the `get` and `set` of the cell! At run-time, the `Cell` will be almost entirely compiled away, so this becomes pretty much equivalent to the version we wrote in the previous part.

```
fn demo_cell(c: &mut Callbacks) {
    {
        let count = Cell::new(0);

        c.register(move |val| {

            let new_count = count.get()+1;
            count.set(new_count);
            println!("Callback 2: {} ({}. time)", val, new_count);
        } );
    }

    c.call(2); c.clone().call(3);
}
```

It is worth mentioning that `Rc` itself also has to make use of interior mutability: When you `clone` an `Rc`, all it has available is a shared reference. However, it has to increment the reference count! Internally, `Rc` uses `Cell` for the count, such that it can be updated during `clone`.

Putting it all together, the story around mutation and ownership through references looks as follows: There are *unique* references, which - because of their exclusivity - are always safe to mutate through. And there are *shared* references, where the compiler cannot generally promise that mutation is safe. However, if extra circumstances guarantee that mutation *is* safe, then it can happen even through a shared reference - as we saw with `Cell`.

RefCell

As the next step in the evolution of `Callbacks`, we could try to solve this problem of mutability once and for all, by adding `Cell` to `Callbacks` such that clients don't have to worry about this. However, that won't end up working: Remember that `Cell` only works with types that are `Copy`, which the environment of a closure will never be. We need a variant of `Cell` that allows borrowing its contents, such that we can provide a `FnMut` with its environment. But if `Cell` would allow that, we could write down all those crashing C++ programs that we wanted to get rid of.

This is the point where our program got too complex for Rust to guarantee at compile-time that nothing bad will happen. Since we don't want to give up the safety guarantee, we are going to need some code that actually checks at run-time that the borrowing rules are not violated. Such a check is provided by `RefCell<T>`: Unlike `Cell<T>`, this lets us borrow the contents, and it works for non-`Copy` `T`. But, as we will see, it incurs some run-time overhead.

Our final version of `Callbacks` puts the closure environment into a `RefCell`.

```
#[derive(Clone)]
struct CallbacksMut {
    callbacks: Vec<Rc<RefCell<FnMut(i32)>>>,
}

impl CallbacksMut {
    pub fn new() -> Self {
        CallbacksMut { callbacks: Vec::new() }
    }

    pub fn register<F: FnMut(i32)+'static>(&mut self, callback: F) {
        let cell = Rc::new(RefCell::new(callback));
        self.callbacks.push(cell);
    }

    pub fn call(&mut self, val: i32) {
        for callback in self.callbacks.iter() {

            let mut closure = callback.borrow_mut();

            (&mut *closure)(val);
        }
    }
}

fn demo_mut(c: &mut CallbacksMut) {
    c.register(|val| println!("Callback 1: {}", val));
    c.call(0);

    {
        let mut count: usize = 0;
        c.register(move |val| {
            count = count+1;
            println!("Callback 2: {} ({}. time)", val, count);
        } );
    }
    c.call(1); c.clone().call(2);
}
```

We have to *explicitly* borrow the contents of a `RefCell` by calling `borrow` or `borrow_mut`. At run-time, the cell will keep track of the number of outstanding shared and mutable references, and panic if the rules are violated.
For this check to be performed, `closure` is a *guard*. Rather than a normal reference, `borrow_mut` returns a smart pointer (`RefMut`, in this case) that waits until is goes out of scope, and then appropriately updates the number of active references.

Since `call` is the only place that borrows the environments of the closures, we should expect that the check will always succeed, as is actually entirely useless. However, this is not actually true. Several different `CallbacksMut` could share a callback (as they were created with `clone`), and calling one callback here could trigger calling all callbacks of the other `CallbacksMut`, which would end up calling the initial callback again. This issue of functions accidentally recursively calling themselves is called *reentrancy*, and it can lead to subtle bugs. Here, it would mean that the closure runs twice, each time thinking it has a unique, mutable reference to its environment - so it may end up dereferencing a dangling pointer. Ouch! Lucky enough, Rust detects this at run-time and panics once we try to borrow the same environment again. I hope this also makes it clear that there's absolutely no hope of Rust performing these checks statically, at compile-time: It would have to detect reentrancy!

Unfortunately, Rust's auto-dereference of pointers is not clever enough here. We thus have to explicitly dereference the smart pointer and obtain a mutable reference to the content.

Now we can repeat the demo from the previous part - but this time, our `CallbacksMut` type can be cloned.

Exercise 12.1: Write some piece of code using only the available, public interface of `CallbacksMut` such that a reentrant call to a closure is happening, and the program panics because the `RefCell` refuses to hand out a second mutable borrow of the closure's environment.