#

Rust-101, Part 11: Trait Objects, Box, Lifetime bounds

We will play around with closures a bit more. Let us implement some kind of generic "callback" mechanism, providing two functions: Registering a new callback, and calling all registered callbacks.

First of all, we need to find a way to store the callbacks. Clearly, there will be a `Vec` involved, so that we can always grow the number of registered callbacks. A callback will be a closure, i.e., something implementing `FnMut(i32)` (we want to call this multiple times, so clearly `FnOnce` would be no good). So our first attempt may be the following. For now, we just decide that the callbacks have an argument of type `i32`.

However, this will not work. Remember how the "type" of a closure is specific to the environment of captured variables. Different closures all implementing `FnMut(i32)` may have different types. However, a `Vec<F>` is a *uniformly typed* vector.

We will thus need a way to store things of *different* types in the same vector. We know all these types implement `FnMut(i32)`. For this scenario, Rust provides *trait objects*: The truth is, `FnMut(i32)` is not just a trait. It is also a type, that can be given to anything implementing this trait. So, we may write the following.

But, Rust complains about this definition. It says something about "Sized". What's the trouble? See, for many things we want to do, it is crucial that Rust knows the precise, fixed size of the type - that is, how large this type will be when represented in memory. For example, for a `Vec`, the elements are stored one right after the other. How should that be possible, without a fixed size? The point is, `FnMut(i32)` could be of any size. We don't know how large that "type that implements `FnMut(i32)`" is. Rust calls this an *unsized* type. Whenever we introduce a type variable, Rust will implicitly add a bound to that variable, demanding that it is sized. That's why we did not have to worry about this so far. You can opt-out of this implicit bound by saying `T: ?Sized`. Then `T` may or may not be sized.

So, what can we do, if we can't store the callbacks in a vector? We can put them in a box. Semantically, `Box<T>` is a lot like `T`: You fully own the data stored there. On the machine, however, `Box<T>` is a *pointer* to a heap-allocated `T`. It is a lot like `std::unique_ptr` in C++. In our current example, the important bit is that since it's a pointer, `T` can be unsized, but `Box<T>` itself will always be sized. So we can put it in a `Vec`.

Now we can provide some functions. The constructor should be straight-forward.

Registration simply stores the callback.

We can also write a generic version of `register`, such that it will be instantiated with some concrete closure type `F` and do the creation of the `Box` and the conversion from `F` to `FnMut(i32)` itself.

For this to work, we need to demand that the type `F` does not contain any short-lived references. After all, we will store it in our list of callbacks indefinitely. If the closure contained a pointer to our caller's stackframe, that pointer could be invalid by the time the closure is called. We can mitigate this by bounding `F` by a *lifetime*: `F: 'a` says that all data of type `F` will *outlive* (i.e., will be valid for at least as long as) lifetime `'a`. Here, we use the special lifetime `'static`, which is the lifetime of the entire program. The same bound has been implicitly added in the version of `register` above, and in the definition of `Callbacks`.

And here we call all the stored callbacks.

Since they are of type `FnMut`, we need to mutably iterate.

Here, `callback` has type `&mut Box<FnMut(i32)>`. We can make use of the fact that `Box` is a *smart pointer*. In particular, we can use it as if it were a normal reference, and use `*` to get to its contents. Then we obtain a mutable reference to these contents, because we call a `FnMut`.

Just like it is the case with normal references, this typically happens implicitly with smart pointers, so we can also directly call the function. Try removing the `&mut *`.

The difference to a reference is that `Box` implies full ownership: Once you drop the box (i.e., when the entire `Callbacks` instance is dropped), the content it points to on the heap will be deleted.

Now we are ready for the demo. Remember to edit `main.rs` to run it.

We can even register callbacks that modify their environment. Per default, Rust will attempt to capture a reference to `count`, to borrow it. However, that doesn't work out this time. Remember the `'static` bound above? Borrowing `count` in the environment would violate that bound, as the reference is only valid for this block. If the callbacks are triggered later, we'd be in trouble. We have to explicitly tell Rust to `move` ownership of the variable into the closure. Its environment will then contain a `usize` rather than a `&mut usize`, and the closure has no effect on this local variable anymore.

Run-time behavior

When you run the program above, how does Rust know what to do with the callbacks? Since an unsized type lacks some information, a *pointer* to such a type (be it a `Box` or a reference) will need to complete this information. We say that pointers to trait objects are *fat*. They store not only the address of the object, but (in the case of trait objects) also a *vtable*: A table of function pointers, determining the code that's run when a trait method is called. There are some restrictions for traits to be usable as trait objects. This is called *object safety* and described in the documentation and the reference. In case of the `FnMut` trait, there's only a single action to be performed: Calling the closure. You can thus think of a pointer to `FnMut` as a pointer to the code, and a pointer to the environment. This is how Rust recovers the typical encoding of closures as a special case of a more general concept.

Whenever you write a generic function, you have a choice: You can make it generic, like `register_generic`. Or you can use trait objects, like `register`. The latter will result in only a single compiled version (rather than one version per type it is instantiated with). This makes for smaller code, but you pay the overhead of the virtual function calls. (Of course, in the case of `register` above, there's no function called on the trait object.) Isn't it beautiful how traits can nicely handle this tradeoff (and much more, as we saw, like closures and operator overloading)?

Exercise 11.1: We made the arbitrary choice of using `i32` for the arguments. Generalize the data structures above to work with an arbitrary type `T` that's passed to the callbacks. Since you need to call multiple callbacks with the same `val: T` (in our `call` function), you will either have to restrict `T` to `Copy` types, or pass a reference.


```
struct CallbacksV1<F: FnMut(i32)> {
    callbacks: Vec<F>,
}
```

```
/* struct CallbacksV2 {
    callbacks: Vec<FnMut(i32)>,
} */
```

```
pub struct Callbacks {
    callbacks: Vec<Box<FnMut(i32)>>,
}

impl Callbacks {
```

```
    pub fn new() -> Self {
        Callbacks { callbacks: Vec::new() }
    }
    pub fn register(&mut self, callback: Box<FnMut(i32)>) {
        self.callbacks.push(callback);
    }
```

```
    pub fn register_generic<F: FnMut(i32)+'static>(&mut self, callback: F) {
        self.callbacks.push(Box::new(callback));
    }
```

```
    pub fn call(&mut self, val: i32) {

        for callback in self.callbacks.iter_mut() {

            (&mut *callback)(val);

        }
    }
}
```

```
pub fn main() {
    let mut c = Callbacks::new();
    c.register(Box::new(|val| println!("Callback 1: {}", val)));
    c.call(0);

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