PREV
13. Utility Traits

NEXT
15. Iterators

Chapter 14. Closures

Save the environment! Create a closure today!

— Cormac Flanagan

Sorting a vector of integers is easy.

```
integers.sort();
```

It is, therefore, a sad fact that when we want some data sorted, it's hardly ever a vector of integers. We typically have records of some kind, and the built-in `sort` method typically does not work:

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}
```

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort();  // error: how do you want them sorted?
}
```

Rust complains that `City` does not implement `std::cmp::Ord`. We need to specify the sort order, like this:

```
/// Helper function for sorting cities by population.
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending);  // ok
}
```

The helper function, `city_population_descending`, takes a `City` record and extracts the *key*, the field by which we want to sort our data. (It returns a negative number because `sort` arranges numbers in increasing order, and we want decreasing order: the most populous city first.) The `sort_by_key` method takes this key-function as a parameter.

This works fine, but it's more concise to write the helper function as a *closure*, an anonymous function expression:

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

The closure here is `|city| -city.population`. It takes an argument `city` and returns `-city.population`. Rust infers the argument type and return type from how the closure is used.

Other examples of standard library features that accept closures include:

- `Iterator` methods such as `map` and `filter`, for working with sequential data. We'll cover these methods in Chapter 15.
- Threading APIs like `thread::spawn`, which starts a new system thread. Concurrency is all about moving work to other threads, and closures conveniently represent units of work. We'll cover these features in Chapter 19.
- Some methods that conditionally need to compute a default value, like the `or_insert_with` method of `HashMap` entries. This method either gets or creates an entry in a `HashMap`, and it's used

when the default value is expensive to compute. The default value is passed in as a closure that is called only if a new entry must be created.

Of course, anonymous functions are everywhere these days, even in languages like Java, C#, Python, and C++ that didn't originally have them. From now on we'll assume you've seen anonymous functions before and focus on what makes Rust's closures a little different. In this chapter, you'll learn the three types of closures, how to use closures with standard library methods, how a closure can "capture" variables in its scope, how to write your own functions and methods that take closures as arguments, and how to store closures for later use as callbacks. We'll also explain how Rust closures are implemented and why they're faster than you might expect.

Capturing Variables

A closure can use data that belongs to an enclosing function. For example:

```
/// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

The closure here uses `stat`, which is owned by the enclosing function, `sort_by_statistic`. We say that the closure "captures" `stat`. This is one of the classic features of closures, so naturally, Rust supports it; but in Rust, this feature comes with a string attached.

In most languages with closures, garbage collection plays an important role. For example, consider this JavaScript code:

```
// Start an animation that rearranges the rows in a table of cities.
function startSortingAnimation(cities, stat) {
    // Helper function that we'll use to sort the table.
    // Note that this function refers to stat.
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    // Now kick off an animation, passing keyfn to it.
    // The sorting algorithm will call keyfn later.
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

The closure `keyfn` is stored in the new `SortingAnimation` object. It's meant to be called after `startSortingAnimation` returns. Now, normally when a function returns, all its variables and arguments go out of scope and are discarded. But here, the JavaScript engine must keep `stat` around somehow, since the closure uses it. Most JavaScript engines do this by allocating `stat` in the heap and letting the garbage collector reclaim it later.

Rust doesn't have garbage collection. How will this work? To answer this question, we'll look at two examples.

Closures That Borrow

First, let's repeat the opening example of this section:

```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

In this case, when Rust creates the closure, it automatically borrows a reference to `stat`. It stands to reason: the closure refers to `stat`, so it must have a reference to it.

The rest is simple. The closure is subject to the rules about borrowing and lifetimes that we described in Chapter 5. In particular, since the closure contains a reference to `stat`, Rust won't let it outlive `stat`. Since the closure is only used during sorting, this example is fine.

In short, Rust ensures safety by using lifetimes instead of garbage collection. Rust's way is faster: even a fast GC allocation will be slower than storing `stat` on the stack, as Rust does in this case.

Closures That Steal

The second example is trickier:

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
```

```
}
```

This is a bit more like what our JavaScript example was doing: `thread::spawn` takes a closure and calls it in a new system thread. Note that `||` is the closure's empty argument list.

The new thread runs in parallel with the caller. When the closure returns, the new thread exits. (The closure's return value is sent back to the calling thread as a `JoinHandle` value. We'll cover that in Chapter 19.)

Again, the closure `key_fn` contains a reference to `stat`. But this time, Rust can't guarantee that the reference is used safely. Rust therefore rejects this program:

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,
              which is owned by the current function
  --> closures_sort_thread.rs:33:18
   |
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
   |              ^^^^^^^^^^^^^^^^^^^^                       ^^^^
   |              |                                     `stat` is borrowed here
   |              may outlive borrowed value `stat`
```

In fact, there are two problems here, because `cities` is shared unsafely as well. Quite simply, the new thread created by `thread::spawn` can't be expected to finish its work before `cities` and `stat` are destroyed at the end of the function.

The solution to both problems is the same: tell Rust to *move* `cities` and `stat` into the closures that use them instead of borrowing references to them.

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

The only thing we've changed is to add the `move` keyword before each of the two closures. The `move` keyword tells Rust that a closure doesn't borrow the variables it uses: it steals them.

The first closure, `key_fn`, takes ownership of `stat`. Then the second closure takes ownership of both `cities` and `key_fn`.

Rust thus offers two ways for closures to get data from enclosing scopes: moves and borrowing. Really there is nothing more to say than that; closures follow the same rules about moves and borrowing that we already covered in Chapters 4 and 5. A few cases in point:

- Just as everywhere else in the language, if a closure would `move` a value of a copyable type, like `i32`, it copies the value instead. So if `Statistic` happened to be a copyable type, we could keep using `stat` even after creating a `move` closure that uses it.

- Values of non-copyable types, like `Vec<City>`, really are moved: the code above transfers `cities` to the new thread, by way of the `move` closure. Rust would not let us access `cities` by name after creating the closure.

- As it happens, this code doesn't need to use `cities` after the point where the closure moves it. If we did, though, the workaround would be easy: we could tell Rust to clone `cities` and store the copy in a different variable. The closure would only steal one of the copies—whichever one it refers to.

We get something important by accepting Rust's strict rules: thread safety. It is precisely because the vector is moved, rather than being shared across threads, that we know the old thread won't free the vector while the new thread is modifying it.

Function and Closure Types

Throughout this chapter, we've seen functions and closures used as values. Naturally, this means that they have types. For example:

```
fn city_population_descending(city: &City) -> i64 {
    -city.population
}
```

This function takes one argument (a `&City`) and returns an `i64`. It has the type `fn(&City) -> i64`.

You can do all the same things with functions that you do with other values. You can store them in variables. You can use all the usual Rust syntax to compute function values:

```
let my_key_fn: fn(&City) -> i64 =
    if user.prefs.by_population {
        city_population_descending
    } else {
        city_monster_attack_risk_descending
```

```
    };

cities.sort_by_key(my_key_fn);
```

Structs may have function-typed fields. Generic types like `Vec` can store scads of functions, as long as they all share the same `fn` type. And function values are tiny: a `fn` value is the memory address of the function's machine code, just like a function pointer in C++.

A function can take another function as an argument. For example:

```
/// Given a list of cities and a test function,
/// return how many cities pass the test.
fn count_selected_cities(cities: &Vec<City>,
                         test_fn: fn(&City) -> bool) -> usize
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}

/// An example of a test function. Note that the type of
/// this function is `fn(&City) -> bool`, the same as
/// the `test_fn` argument to `count_selected_cities`.
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// How many cities are at risk for monster attack?
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

If you're familiar with function pointers in C/C++, you'll see that Rust's function values are exactly the same thing.

After all this, it may come as a surprise that closures do *not* have the same type as functions:

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // error: type mismatch
```

The second argument causes a type error. To support closures, we must change the type signature of this function. It needs to look like this:

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

We have changed only the type signature of `count_selected_cities`, not the body. The new version is generic. It takes a `test_fn` of any type `F` as long as `F` implements the special trait `Fn(&City) -> bool`. This trait is automatically implemented by all functions and most closures that take a single `&City` as an argument and return a Boolean value.

```
fn(&City) -> bool    // fn type (functions only)
Fn(&City) -> bool    // Fn trait (both functions and closures)
```

This special syntax is built into the language. The `->` and return type are optional; if omitted, the return type is `()`.

The new version of `count_selected_cities` accepts either a function or a closure:

```
count_selected_cities(
    &my_cities,
    has_monster_attacks);  // ok

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit);  // also ok
```

Why didn't our first attempt work? Well, a closure is callable, but it's not a `fn`. The closure `|city| city.monster_attack_risk > limit` has its own type that's not a `fn` type.

In fact, every closure you write has its own type, because a closure may contain data: values either borrowed or stolen from enclosing scopes. This could be any number of variables, in any combination of types. So every closure has an ad hoc type created by the compiler, large enough to hold that data. No two

closures have exactly the same type. But every closure implements a `Fn` trait; the closure in our example implements `Fn(&City) -> i64`.

Since every closure has its own type, code that works with closures usually needs to be generic, like `count_selected_cities`. It's a little clunky to spell out the generic types each time, but to see the advantages of this design, just read on.

Closure Performance

Rust's closures are designed to be fast: faster than function pointers, fast enough that you can use them even in red-hot, performance-sensitive code. If you're familiar with C++ lambdas, you'll find that Rust closures are just as fast and compact, but safer.

In most languages, closures are allocated in the heap, dynamically dispatched, and garbage collected. So creating them, calling them, and collecting them each cost a tiny bit of extra CPU time. Worse, closures tend to rule out *inlining*, a key technique compilers use to eliminate function call overhead and enable a raft of other optimizations. All told, closures are slow enough in these languages that it can be worth manually removing them from tight inner loops.

Rust closures have none of these performance drawbacks. They're not garbage collected. Like everything else in Rust, they aren't allocated on the heap unless you put them in a `Box`, `Vec`, or other container. And since each closure has a distinct type, whenever the Rust compiler knows the type of the closure you're calling, it can inline the code for that particular closure. This makes it OK to use closures in tight loops, and Rust programs often do so, enthusiastically, as you'll see in Chapter 15.

Figure 14-1 shows how Rust closures are laid out in memory. At the top of the figure, we show a couple of local variables that our closures will refer to: a string `food` and a simple enum `weather`, whose numeric value happens to be 27.

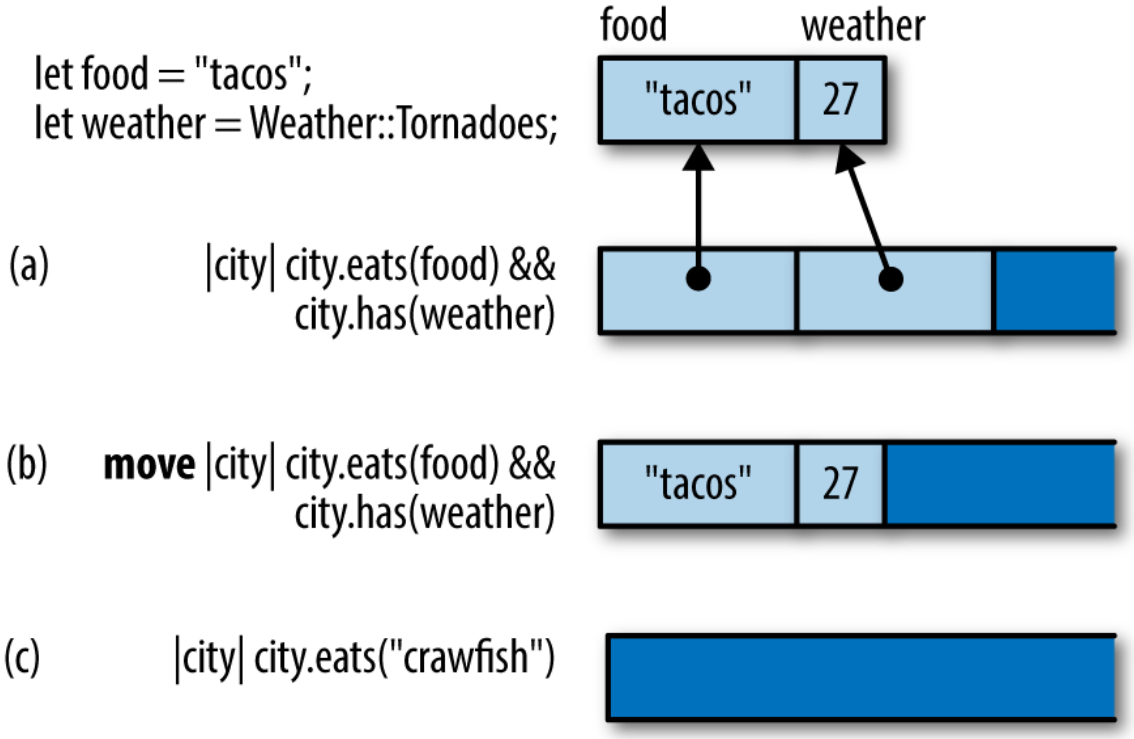

Figure 14-1. Layout of closures in memory

Closure (a) uses both variables. Apparently we're looking for cities that have both tacos and tornadoes. In memory, this closure looks like a small struct containing references to the variables it uses.

Note that it doesn't contain a pointer to its code! That's not necessary: as long as Rust knows the closure's type, it knows which code to run when you call it.

Closure (b) is exactly the same, except it's a `move` closure, so it contains values instead of references.

Closure (c) doesn't use any variables from its environment. The struct is empty, so this closure does not take up any memory at all.

As the figure shows, these closures don't take up much space. But even those few bytes are not always needed in practice. Often, the compiler can inline all calls to a closure, and then even the small structs shown in this figure are optimized away.

In "Callbacks", we'll show how to allocate closures in the heap and call them dynamically, using trait objects. That is a bit slower, but it is still as fast as any other trait object method.

Closures and Safety

Throughout the chapter so far, we've talked about how Rust ensures that closures respect the language's safety rules when they borrow or move variables from the surrounding code. But there are some further

consequences that are not exactly obvious. In this section, we'll explain a bit more about what happens when a closure drops or modifies a captured value.

Closures That Kill

We have seen closures that borrow values and closures that steal them; it was only a matter of time before they went all the way bad.

Of course, *kill* is not really the right terminology. In Rust, we *drop* values. The most straightforward way to do it is to call `drop()`:

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

When `f` is called, `my_str` is dropped.

So what happens if we call it twice?

```
f();
f();
```

Let's think it through. The first time we call `f`, it drops `my_str`, which means the memory where the string is stored is freed, returned to the system. The second time we call `f`, the same thing happens. It's a *double free*, a classic mistake in C++ programming that triggers undefined behavior.

Dropping a `String` twice would be an equally bad idea in Rust. Fortunately, Rust can't be fooled so easily:

```
f();  // ok
f();  // error: use of moved value
```

Rust knows this closure can't be called twice.

A closure that can only be called once may seem like a rather extraordinary thing, but we've been talking throughout this book about ownership and lifetimes. The idea of values being used up (that is, moved) is one of the core concepts in Rust. It works the same with closures as with everything else.

FnOnce

Let's try once more to trick Rust into dropping a `String` twice. This time, we'll use this generic function:

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

This generic function may be passed any closure that implements the trait `Fn()`: that is, closures that take no arguments and return `()`. (As with functions, the return type can be omitted if it's `()`; `Fn()` is shorthand for `Fn() -> ()`.)

Now what happens if we pass our unsafe closure to this generic function?

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

Again, the closure will drop `my_str` when it's called. Calling it twice would be a double free. But again, Rust is not fooled:

```
error[E0525]: expected a closure that implements the `Fn` trait, but
              this closure only implements `FnOnce`

  --> closures_twice.rs:12:13
    |
  8 | let f = || drop(my_str);
    |         ^^^^^^^^------^
    |         |       |
    |         |       closure is `FnOnce` because it moves the variable `my_str`
    |         |       out of its environment
    |         this closure implements `FnOnce`, not `Fn`
  9 | call_twice(f);
    | ---------- the requirement to implement `Fn` derives from here
```

This error message tells us more about how Rust handles "closures that kill." They could have been banned from the language entirely, but cleanup closures are useful sometimes. So instead, Rust restricts their use. Closures that drop values, like `f`, are not allowed to have `Fn`. They are, quite literally, no `Fn` at all. They implement a less powerful trait, `FnOnce`, the trait of closures that can be called once.

The first time you call a `FnOnce` closure, *the closure itself is used up.* It's as though the two traits, `Fn` and `FnOnce`, were defined like this:

```
// Pseudocode for `Fn` and `FnOnce` traits with no arguments.
trait Fn() -> R {
    fn call(&self) -> R;
}
```

```
trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Just as an arithmetic expression like `a + b` is shorthand for a method call, `Add::add(a, b)`, Rust treats `closure()` as shorthand for one of the two trait methods shown above. For a `Fn` closure, `closure()` expands to `closure.call()`. This method takes `self` by reference, so the closure is not moved. But if the closure is only safe to call once, then `closure()` expands to `closure.call_once()`. That method takes `self` by value, so the closure is used up.

Of course we've been deliberately stirring up trouble here by using `drop()`. In practice, you'll mostly get into this situation by accident. It doesn't happen often, but once in a great while you'll write some closure code that unintentionally uses up a value:

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict {  // oops!
        println!("{:?} - {:?}", key, value);
    }
};
```

Then, when you call `debug_dump_dict()` more than once, you'll get an error message like this:

```
error[E0382]: use of moved value: `debug_dump_dict`
  --> closures_debug_dump_dict.rs:18:5
   |
19 |     debug_dump_dict();
   |     ----------------- `debug_dump_dict` moved due to this call
20 |     debug_dump_dict();
   |     ^^^^^^^^^^^^^^^ value used here after move
   |
note: closure cannot be invoked more than once because it moves the variable
`dict` out of its environment
  --> src/main.rs:13:29
   |
13 |         for (key, value) in dict {
   |                             ^^^^
```

To debug this, we have to figure out why the closure is a `FnOnce`. Which value is being used up here? The compiler helpfully points out that it's `dict`, which in this case is the only one we're referring to at all. Ah, there's the bug: we're using up `dict` by iterating over it directly. We should be looping over `&dict` rather than plain `dict`, to access the values by reference:

```
let debug_dump_dict = || {
    for (key, value) in &dict {  // does not use up dict
        println!("{:?} - {:?}", key, value);
    }
};
```

This fixes the error; the function is now a `Fn` and can be called any number of times.

FnMut

There is one more kind of closure, the kind that contains mutable data or `mut` references.

Rust considers non-`mut` values safe to share across threads. But it wouldn't be safe to share non-`mut` closures that contain `mut` data: calling such a closure from multiple threads could lead to all sorts of race conditions as multiple threads try to read and write the same data at the same time.

Therefore, Rust has one more category of closure, `FnMut`, the category of closures that write. `FnMut` closures are called by `mut` reference, as if they were defined like this:

```
// Pseudocode for `Fn`, `FnMut`, and `FnOnce` traits.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Any closure that requires `mut` access to a value, but doesn't drop any values, is a `FnMut` closure. For example:

```
let mut i = 0;
let incr = || {
    i += 1;  // incr borrows a mut reference to i
    println!("Ding! i is now: {}", i);
};
call_twice(incr);
```

The way we wrote `call_twice`, it requires a `Fn`. Since `incr` is a `FnMut` and not a `Fn`, this code fails to compile. There's an easy fix, though. To understand the fix, let's take a step back and summarize what

you've learned about the three categories of Rust closures.

- `Fn` is the family of closures and functions that you can call multiple times without restriction. This highest category also includes all `fn` functions.
- `FnMut` is the family of closures that can be called multiple times if the closure itself is declared `mut`.
- `FnOnce` is the family of closures that can be called once, if the caller owns the closure.

Every `Fn` meets the requirements for `FnMut`, and every `FnMut` meets the requirements for `FnOnce`. As shown in Figure 14-2, they're not three separate categories.

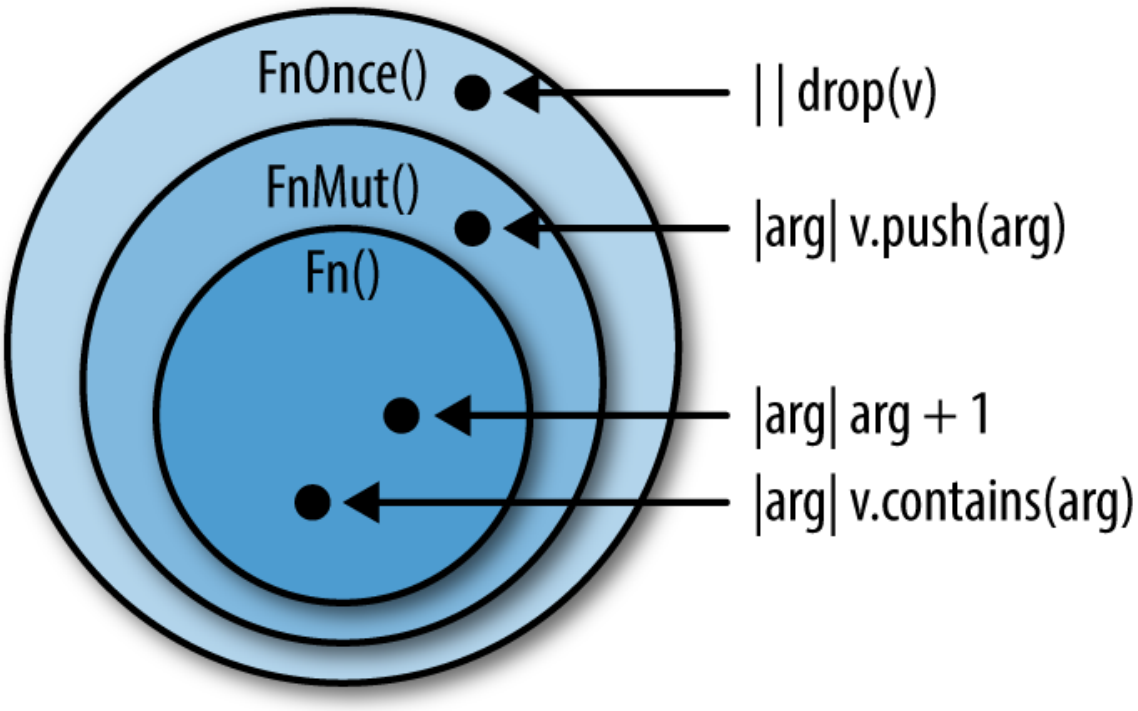

Figure 14-2. Venn diagram of the three closure categories

Instead, `Fn()` is a subtrait of `FnMut()`, which is a subtrait of `FnOnce()`. This makes `Fn` the most exclusive and most powerful category. `FnMut` and `FnOnce` are broader categories that include closures with usage restrictions.

Now that we've organized what we know, it's clear that to accept the widest possible swath of closures, our `call_twice` function really ought to accept all `FnMut` closures, like this:

```
fn call_twice<F>(mut closure: F) where F: FnMut() {
    closure();
    closure();
}
```

The bound on the first line was `F: Fn()`, and now it's `F: FnMut()`. With this change, we still accept all `Fn` closures, and we additionally can use `call_twice` on closures that mutate data:

```
let mut i = 0;
call_twice(|| i += 1);  // ok!
assert_eq!(i, 2);
```

Copy and Clone for Closures

Just as Rust automatically figures out which closures can be called only once, it can figure out which closures can implement `Copy` and `Clone`, and which cannot.

As we explained earlier, closures are represented as structs which contain either the values (for `move` closures) or references to the values (for non-`move` closures) of the variables they capture. The rules for `Copy` and `Clone` on closures are just like the `Copy` and `Clone` rules for regular structs. A non-`move` closure that doesn't mutate variables holds only shared references, which are both `Clone` and `Copy`, so that closure is both `Clone` and `Copy` as well:

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y;                 // This closure is `Copy`, so...
assert_eq!(add_y(copy_of_add_y(22)), 42); // ... we can call both.
```

On the other hand, a non-`move` closure that *does* mutate values has mutable references within its internal representation. Mutable references are neither `Clone` nor `Copy`, so neither is a closure that uses them:

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x;          // this moves, rather than copies
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // error: use of moved value
```

For a `move` closure, the rules are even simpler. If everything a `move` closure captures is `Copy`, it's `Copy`. If everything it captures is `Clone`, it's `Clone`. For instance:

```
let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{}", greeting);
};
```

```
greet.clone()("Alfred");
greet.clone()("Bruce");
```

This `.clone()(...)` syntax is a little weird, but it just means that we clone the closure, and then call the clone. This program outputs:

```
Hello, Alfred
Hello, Bruce
```

When `greeting` is used in `greet`, it's moved into the struct that represents `greet` internally, because it's a `move` closure. So, when we clone `greet`, everything inside it is cloned, too. There are two copies of `greeting`, which are each modified separately when the clones of `greet` are called. This isn't so useful on its own, but when you need to pass the same closure into more than one function, it can be very helpful.

Callbacks

A lot of libraries use *callbacks* as part of their API: functions provided by the user, for the library to call later. In fact, you've seen some APIs like that already in this book. Back in Chapter 2, we used the Iron framework to write a simple web server. It looked like this:

```
fn main() {
    let mut router = Router::new();

    router.get("/", get_form, "root");
    router.post("/gcd", post_gcd, "gcd");

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

The purpose of the router is to route incoming requests from the Internet to the bit of Rust code that handles that particular kind of request. In this example, `get_form` and `post_gcd` were the names of some functions that we declared elsewhere in the program, using the `fn` keyword. But we could have passed closures instead, like this:

```
let mut router = Router::new();

router.get("/", |_: &mut Request| {
    Ok(get_form_response())
}, "root");
router.post("/gcd", |request: &mut Request| {
    let numbers = get_numbers(request)?;
```

```
        Ok(get_gcd_response(numbers))
    }, "gcd");
```

This is because Iron was written to accept any thread-safe `Fn` as an argument.

How can we do that in our own programs? Let's try writing our own very simple router from scratch, without using any code from Iron. We can begin by declaring a few types to represent HTTP requests and responses:

```
struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

Now the job of a router is simply to store a table that maps URLs to callbacks, so that the right callback can be called on demand. (For simplicity's sake, we'll only allow users to create routes that match a single exact URL.)

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// Create an empty router.
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// Add a route to the router.
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}
```

Unfortunately, we've made a mistake. Did you notice it?

This router works fine as long as we only add one route to it:

```
let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());
```

This much compiles and runs. Unfortunately, if we add another route:

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

then we get errors:

```
error[E0308]: mismatched types
  --> closures_bad_router.rs:41:30
   |
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                              expected closure, found a different closure
   |
   = note: expected type `[closure@closures_bad_router.rs:40:27: 40:50]`
              found type `[closure@closures_bad_router.rs:41:30: 41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait object
```

Our mistake was in how we defined the `BasicRouter` type:

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

We unwittingly declared that each `BasicRouter` has a single callback type `C`, and all the callbacks in the `HashMap` are of that type. Back in "Which to Use", we showed a `Salad` type that had the same problem.

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

The solution here is the same as for the salad: since we want to support a variety of types, we need to use boxes and trait objects.

```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>;

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

Each box can contain a different type of closure, so a single `HashMap` can contain all sorts of callbacks. Note that the type parameter `C` is gone.

This requires a few adjustments to the methods:

```
impl BasicRouter {
    // Create an empty router.
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }

    }

    // Add a route to the router.
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}
```

(Note the two bounds on `C` in the type signature for `add_route`: a particular `Fn` trait, and the `'static` lifetime. Rust makes us add this `'static` bound. Without it, the call to `Box::new(callback)` would be an error, because it's not safe to store a closure if it contains borrowed references to variables that are about to go out of scope.)

Finally, our simple router is ready to handle incoming requests:

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

At the cost of some flexibility, we could also write a more space-efficient version of this router that, rather than storing trait objects, uses *function pointers*, or `fn` types. These types, such as `fn(u32) -> u32`, act a lot like closures.

```
fn add_ten(x: u32) -> u32 {
    x + 10
}
```

```
let fn_ptr: fn(u32) -> u32 = add_ten;
let eleven = fn_ptr(1); //11
```

In fact, closures that don't capture anything from their environment are identical to function pointers, since they don't need to hold any extra information about captured variables. If you specify the appropriate `fn` type, either in a binding or a function signature, the compiler is happy to let you use them that way:

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

Unlike capturing closures, these function pointers take up only a single `usize`.

Function pointers can also be used to make our own implementation of dynamic dispatch, rather than using the compiler's as we were with `Box<dyn Fn()>`.

```
struct FnPointerRouter {
    routes: HashMap<String, fn(&Request) -> Response>
}
```

Here, the `HashMap` is storing just a single `usize` per `String`, and critically, there's no `Box`. Aside from the `HashMap` itself, there's no dynamic allocation at all. Of course, the methods need to be adjusted as well:

```
impl FnPointerRouter {
    // Create an empty router.
    fn new() -> FnPointerRouter {
        FnPointerRouter { routes: HashMap::new() }
    }

    // Add a route to the router.
    fn add_route(&mut self, url: &str, callback: fn(&Request) -> Response)
    {
        self.routes.insert(url.to_string(), callback);
    }
}
```

As laid out in Figure 14-1, closures have unique types because each one captures different variables, so among other things, they're each a different size. If they don't capture anything, though, there's nothing to store. By using `fn` pointers in functions that take callbacks, you can restrict a caller to use only these non-capturing closures, gaining some perfomance and flexibility within the code using callbacks at the cost of flexibility for the users of your API.

Using Closures Effectively

As we've seen, Rust's closures are different from closures in most other languages. The biggest difference is that in languages with GC, you can use local variables in a closure without having to think about lifetimes or ownership. Without GC, things are different. Some design patterns that are commonplace in Java, C#, and JavaScript won't work in Rust without changes.

For example, take the Model-View-Controller design pattern (MVC for short), illustrated in Figure 14-3. For every element of a user interface, an MVC framework creates three objects: a *model* representing that UI element's state, a *view* that's responsible for its appearance, and a *controller* that handles user interaction. Countless variations on MVC have been implemented over the years, but the general idea is that three objects divvy up the UI responsibilities somehow.

Here's the problem. Typically, each object has a reference to one or both of the others, directly or through a callback, as shown in Figure 14-3. Whenever anything happens to one of the objects, it notifies the others, so everything updates promptly. The question of which object "owns" the others never comes up.

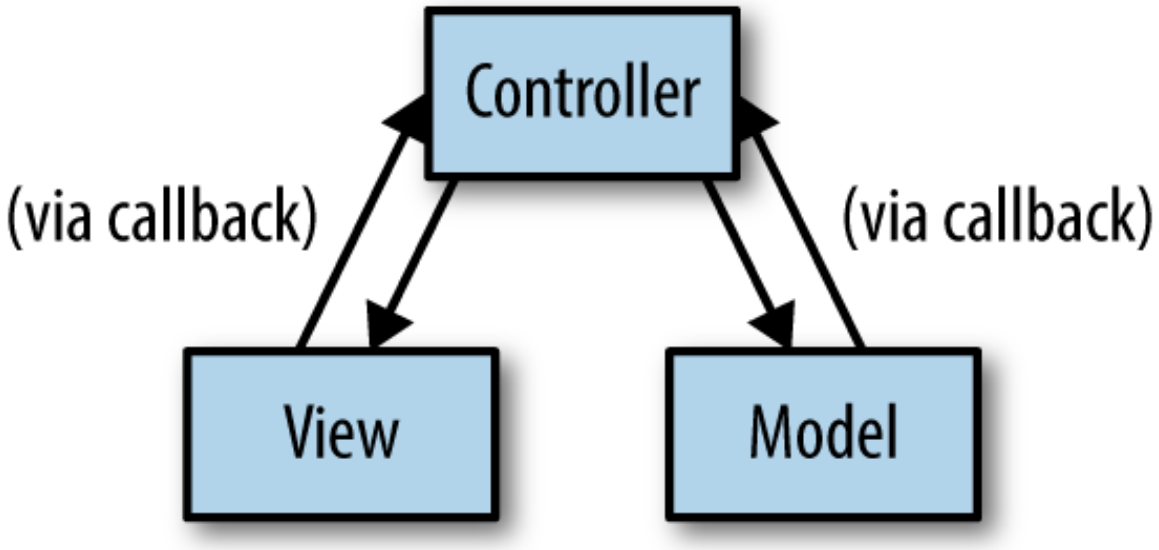

Figure 14-3. The Model-View-Controller design pattern

You can't implement this pattern in Rust without making some changes. Ownership must be made explicit, and reference cycles must be eliminated. The model and the controller can't have direct references to each other.

Rust's radical wager is that good alternative designs exist. Sometimes you can fix a problem with closure ownership and lifetimes by having each closure receive the references it needs as arguments. Sometimes you can assign each thing in the system a number and pass around the numbers instead of references. Or you can implement one of the many variations on MVC where the objects don't all have references to

each other. Or model your toolkit after a non-MVC system with unidirectional data flow, like Facebook's Flux architecture, shown in Figure 14-4.

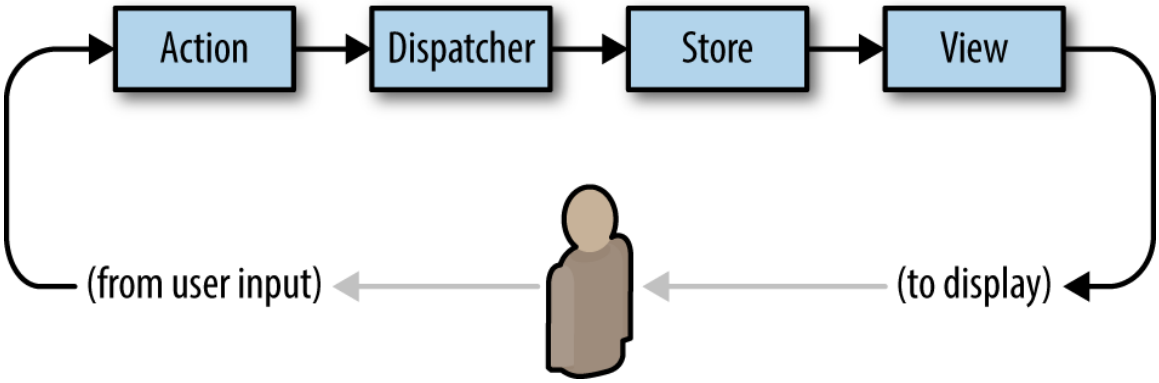

Figure 14-4. The Flux architecture, an alternative to MVC

In short, if you try to use Rust closures to make a "sea of objects," you're going to have a hard time. But there are alternatives. In this case, it seems software engineering as a discipline is already gravitating to the alternatives anyway, because they're simpler.

In the next chapter, we turn to a topic where closures really shine. We'll be writing a kind of code that takes full advantage of the concision, speed, and efficiency of Rust closures and that's fun to write, easy to read, and eminently practical. Up next: Rust iterators.

PREV
13. Utility Traits

NEXT
15. Iterators