Chapter 13

Functional Language features in Rust: Iterators and Closures

Rust’s design has taken inspiration from a lot of existing languages and techniques, and one significant influence is functional programming. Programming in a functional style often includes using functions as values, by passing them in arguments, returning them from other functions, assigning them to variables for later execution, and so forth. We won’t debate here the issue of what, exactly, functional programming is or is not, but will instead show off some features of Rust that are similar to features in many languages often referred to as functional.

More specifically, we’re going to cover:

Closures: a function-like construct you can store in a variable.

Iterators: a way of processing a series of elements.

How to use these features to improve on the I/O project from Chapter 12.

The performance of these features. Spoiler alert: they’re faster than you might think!

There are other Rust features influenced by the functional style, like pattern matching and enums, that we’ve covered in other chapters as well. Mastering closures and iterators is an important part of writing idiomatic, fast Rust code, so we’re devoting an entire chapter to them here.

Closures: Anonymous Functions that can Capture their Environment

Rust’s closures are anonymous functions you can save in a variable or pass as arguments to other functions. You can create the closure in one place, and then call the closure to evaluate it in a different context. Unlike functions, closures are able to capture values from the scope in which they are called. We’re going to demonstrate how these features of closures allow for code reuse and customization of behavior.

Creating an Abstraction of Behavior Using a Closure

Let’s work on an example of a situation in which it’s useful to store a closure to be executed at a later time. We’ll talk about the syntax of closures, type inference, and traits along the way.

The hypothetical situation is this: we work at a startup that’s making an app to generate custom exercise workout plans. The backend is written in Rust, and the algorithm that generates the workout plan takes into account many different factors, like the app user’s age, Body Mass Index, preferences, recent workouts, and an intensity number they specify. The actual algorithm used isn’t important in this example; what’s important is that this calculation takes a few seconds. We only want to call this algorithm when we need to, and only call it once, so we aren’t making the user wait more than necessary.

We’ll simulate calling this hypothetical algorithm with the simulated\_expensive\_calculation function shown in Listing 13-1, which will print calculating slowly..., wait for two seconds, and then return whatever number we passed in:

Filename: src/main.rs

use std::thread;

use std::time::Duration;

fn simulated\_expensive\_calculation(intensity: i32) -> i32 {

println!("calculating slowly...");

thread::sleep(Duration::from\_secs(2));

intensity

}

Listing 13-1: A function to stand in for a hypothetical calculation that takes about two seconds to run

Next, we have a main function that contains the parts of the workout app important for this example. This represents the code that the app would call when a user asks for a workout plan. Because the interaction with the app’s frontend isn’t relevant to the use of closures, we’re going to hardcode values representing inputs to our program and print the outputs.

The required inputs are:

An intensity number from the user, specified when they request a workout to indicate whether they’d like a low intensity workout or a high intensity workout

A random number that will generate some variety in the workout plans

The output will be the recommended workout plan.

Listing 13-2 shows the main function we’re going to use.

Filename: src/main.rs

fn main() {

let simulated\_user\_specified\_value = 10;

let simulated\_random\_number = 7;

generate\_workout(simulated\_user\_specified\_value, simulated\_random\_number);

}

Listing 13-2: A main function with hardcoded values to simulate user input and random number generation

.

There’s the context, so let’s get to the algorithm. The generate\_workout function in Listing 13-3 contains the business logic of the app that we’re most concerned with in this example. The rest of the code changes in this example will be made to this function:

Filename: src/main.rs

fn generate\_workout(intensity: i32, random\_number: i32) {

 if intensity < 25 {

println!(

"Today, do {} pushups!",

simulated\_expensive\_calculation(intensity)

);

println!(

"Next, do {} situps!",

simulated\_expensive\_calculation(intensity)

);

} else {

 if random\_number == 3 {

println!("Take a break today! Remember to stay hydrated!");

 } else {

println!(

"Today, run for {} minutes!",

simulated\_expensive\_calculation(intensity)

)

}

}

}

Listing 13-3: The business logic that prints the workout plans based on the inputs and calls to the simulated\_expensive\_calculation function

The code in Listing 13-3 has multiple calls to the slow calculation function. The first if block calls simulated\_expensive\_calculation twice, the if inside the outer else  doesn’t call it at all, and the code inside the second else case  calls it once.

The desired behavior of the generate\_workout function is to first check if the user wants a low intensity workout (indicated by a number less than 25) or a high intensity workout (25 or more).

Low intensity workout plans will recommend a number of pushups and situps based on the complex algorithm we’re simulating.

If the user wants a high intensity workout, there’s some additional logic: if the value of the random number generated by the app happens to be 3, the app will recommend a break and hydration. If not, the user will get a number of minutes of running based on the complex algorithm.

The data science team has let us know that we’ll have to make some changes to the way we call the algorithm in future. To simplify the update when those changes happen, we want to refactor this code so it only calls the simulated\_expensive\_calculation function once. We also want to cut the place where we’re currently calling the function twice unnecessarily without adding any other calls to that function in the process. That is, we don’t want to call it if the result isn’t needed, and we still want to call it only once.

Refactoring Using Functions

There are many ways we could restructure this program. First we’ll try extracting the duplicated call to the expensive calculation function into a variable, as shown in Listing 13-4:

Filename: src/main.rs

fn generate\_workout(intensity: i32, random\_number: i32) {

let expensive\_result =

simulated\_expensive\_calculation(intensity);

if intensity < 25 {

println!(

"Today, do {} pushups!",

expensive\_result

);

println!(

"Next, do {} situps!",

expensive\_result

);

} else {

if random\_number == 3 {

println!("Take a break today! Remember to stay hydrated!");

} else {

println!(

"Today, run for {} minutes!",

expensive\_result

)

}

}

}

Listing 13-4: Extracting the calls to simulated\_expensive\_calculation to one place and storing the result in the expensive\_result variable

This change unifies all the calls to simulated\_expensive\_calculation and solves the problem of the first if block calling the function twice unnecessarily. Unfortunately, we’re now calling this function and waiting for the result in all cases, which includes the inner if block that doesn’t use the result value at all.

We want to define code in one place in our program, but only execute that code where we actually need the result. This is a use case for closures!

Refactoring with Closures to Store Code for Later Execution

Instead of always calling the simulated\_expensive\_calculation function before the if blocks, we can define a closure and store the closure in a variable rather than storing the result, as shown in Listing 13-5. We can actually move the whole body of simulated\_expensive\_calculation within the closure we’re introducing here:

Filename: src/main.rs

let expensive\_closure = |num| {

println!("calculating slowly...");

thread::sleep(Duration::from\_secs(2));

num

};

Listing 13-5: Defining a closure and storing it in the expensive\_closure variable

The closure definition comes after the = to assign it to the variable expensive\_closure. To define a closure, we start with a pair of vertical pipes (|), inside which we specify the parameters to the closure; this syntax was chosen because of its similarity to closure definitions in Smalltalk and Ruby. This closure has one parameter named num; if we had more than one parameter, we would separate them with commas, like |param1, param2|.

After the parameters, we place curly braces that hold the body of the closure—these are optional is the closure body is a single expression. The end of the closure, after the curly braces, needs a semicolon to complete the let statement. The value returned from the last line in the closure body (num) will be the value returned from the closure when it’s called, since that line doesn’t end in a semicolon; just like in function bodies.

Note that this let statement means expensive\_closure contains the definition of an anonymous function, not the resulting value of calling the anonymous function. Recall that we’re using a closure because we want to define the code to call at one point, store that code, and actually call it at a later point; the code we want to call is now stored in expensive\_closure.

Now that we have the closure defined, we can change the code in the if blocks to call the closure, in order to execute the code and get the resulting value. We call a closure like we do a function: we specify the variable name that holds the closure definition and follow it with parentheses containing the argument values we want to use, as shown in Listing 13-6:

Filename: src/main.rs

fn generate\_workout(intensity: i32, random\_number: i32) {

let expensive\_closure = |num| {

println!("calculating slowly...");

thread::sleep(Duration::from\_secs(2));

num

};

if intensity < 25 {

println!(

"Today, do {} pushups!",

expensive\_closure(intensity)

);

println!(

"Next, do {} situps!",

expensive\_closure(intensity)

);

} else {

if random\_number == 3 {

println!("Take a break today! Remember to stay hydrated!");

} else {

println!(

"Today, run for {} minutes!",

expensive\_closure(intensity)

)

}

}

}

Listing 13-6: Calling the expensive\_closure we’ve defined

Now the expensive calculation is called to only one place, and we’re only executing that code where we need the results.

We have, however, reintroduced one of the problems from Listing 13-3: we’re still calling the closure twice in the first if block, which will call the expensive code twice and make the user wait twice as long as they need to. We could fix this problem by creating a variable local to that if block to hold the result of calling the closure, but closures provide us with another solution. We’ll get back to that solution in a bit; let’s first talk about why there aren’t type annotations in the closure definition and the traits involved with closures.

Closure Type Inference and Annotation

Closures differ from functions defined with the fn keyword in a few ways. The first is that closures don’t require you to annotate the types of the parameters or the return value like fn functions do.

Type annotations are required on functions because they are part of an explicit interface exposed to your users. Defining this interface rigidly is important for ensuring that everyone agrees on what types of values a function uses and returns. Closures aren’t used in an exposed interface like this, though: they’re stored in variables and used without naming them and exposing them to users of our library.

Additionally, closures are usually short and only relevant within a narrow context rather than in any arbitrary scenario. Within these limited contexts, the compiler is reliably able to infer the types of the parameters and return type, similar to how it’s able to infer the types of most variables.

Making users annotate the types in these small, anonymous functions would be annoying and largely redundant with the information the compiler already has available.

Like variables, we can choose to add type annotations if we want to increase explicitness and clarity at the cost of being more verbose than is strictly necessary; annotating the types for the closure we defined in Listing 13-4 would look like the definition shown in Listing 13-7:

Filename: src/main.rs

let expensive\_closure = |num: i32| -> i32 {

println!("calculating slowly...");

thread::sleep(Duration::from\_secs(2));

num

};

Listing 13-7: Adding optional type annotations of the parameter and return value types in the closure

The syntax of closures and functions looks more similar with type annotations. Here’s a vertical comparison of the syntax for the definition of a function that adds one to its parameter, and a closure that has the same behavior. We’ve added some spaces here to line up the relevant parts). This illustrates how closure syntax is similar to function syntax, except for the use of pipes and the amount of syntax that is optional:

fn add\_one\_v1 (x: i32) -> i32 { x + 1 }

let add\_one\_v2 = |x: i32| -> i32 { x + 1 };

let add\_one\_v3 = |x| { x + 1 };

let add\_one\_v4 = |x| x + 1 ;

prod: extra spacing is intentional, pleas retain in layout

The first line shows a function definition, and the second line shows a fully annotated closure definition. The third line removes the type annotations from the closure definition, and the fourth line removes the braces that are optional, since the closure body only has one line. These are all valid definitions that will produce the same behavior when they’re called.

Closure definitions will have one concrete type inferred for each of their parameters and for their return value. For instance, Listing 13-8 shows the definition of a short closure that just returns the value it receives as a parameter.

This closure isn’t very useful except for the purposes of this example. Note that we haven’t added any type annotations to the definition: if we then try to call the closure twice, using a String as an argument the first time and an i32 the second time, we’ll get an error:

Filename: src/main.rs

let example\_closure = |x| x;

let s = example\_closure(String::from("hello"));

let n = example\_closure(5);

Listing 13-8: Attempting to call a closure whose types are inferred with two different types

The compiler gives us this error:

error[E0308]: mismatched types

--> src/main.rs

|

| let n = example\_closure(5);

| ^ expected struct `std::string::String`, found

integral variable

|

= note: expected type `std::string::String`

found type `{integer}`

The first time we call example\_closure with the String value, the compiler infers the type of x and the return type of the closure to be String. Those types are then locked in to the closure in example\_closure, and we get a type error if we try to use a different type with the same closure.

Storing Closures Using Generic Parameters and the Fn Traits

Returning to our workout generation app, in Listing 13-6 we left our code still calling the expensive calculation closure more times than it needs to. One option to solve this issue is to save the result of the expensive closure in a variable for reuse and use the variable instead in each place we need to result instead of calling the closure again. This method, though, could result in a lot of repeated code.

Because we have a closure for the expensive calculation, we have another solution available to us. We can create a struct that will hold the closure and the resulting value of calling the closure. The struct will only execute the closure if we need the resulting value, and it will cache the resulting value so that the rest of our code doesn’t have to be responsible for saving and reusing the result. You may know this pattern as memoization or lazy evaluation.

In order to make a struct that holds a closure, we need to be able to specify the type of the closure, because a struct definition needs to know the types of each of its fields. Each closure instance has its own unique anonymous type: that is, even if two closures have the same signature, their types are still considered different. In order to define structs, enums, or function parameters that use closures, we use generics and trait bounds like we discussed in Chapter 10.

The Fn traits are provided by the standard library. All closures implement one of the traits Fn, FnMut, or FnOnce. We’ll discuss the difference between these traits in the next section on capturing the environment; in this example, we can use the Fn trait.

We add types to the Fn trait bound to represent the types of the parameters and return values the closures must have in order to match this trait bound. In this case, our closure has a parameter of type i32 and returns an i32, so the trait bound we specify is Fn(i32) -> i32.

Listing 13-9 shows the definition of the Cacher struct that holds a closure and an optional result value:

Filename: src/main.rs

struct Cacher<T>

where T: Fn(i32) -> i32

{

calculation: T,

value: Option<i32>,

}

Listing 13-9: Defining a Cacher struct that holds a closure in calculation and an optional result in value

The Cacher struct has a calculation field of the generic type T. The trait bounds on T specify that it’s a closure using the Fn trait. Any closure we want to store in the calculation field instance must have one i32 parameter (specified within the parentheses after Fn) and must return an i32 (specified after the ->).

The value field is of type Option<i32>. Before we execute the closure, value will be None. When the code asks for the result of the closure, it will execute the closure at that time and store the result within a Some variant in the value field. Then if the code asks for the result of the closure again, instead of executing the closure again, it’ll return the result held in the Some variant.

The logic around the value field we’ve just described is defined in Listing 13-10:

Filename: src/main.rs

impl<T> Cacher<T>

where T: Fn(i32) -> i32

{

fn new(calculation: T) -> Cacher<T> {

Cacher {

calculation,

value: None,

}

}

fn value(&mut self, arg: i32) -> i32 {

match self.value {

Some(v) => v,

None => {

let v = (self.calculation)(arg);

self.value = Some(v);

v

},

}

}

}

Listing 13-10: The caching logic of Cacher

We want Cacher to manage the struct fields’ values, rather than letting the calling code potentially change the values in these fields directly, so these fields are private.

The Cacher::new function takes a generic parameter T, which we’ve defined as having the same trait bound as the Cacher struct. Then Cacher::new returns a Cacher instance that holds the closure specified in the calculation field and a None value in the value field, since we haven’t executed the closure yet.

When the calling code wants the result of evaluating the closure, instead of calling the closure directly, it will call the value method. This method checks to see if we already have a resulting value in self.value in a Some; if we do, it returns the value within the Some without executing the closure again.

If self.value is None, we call the closure stored in self.calculation, save the result in self.value for future use, and return the value as well.

Listing 13-11 shows how we can use this Cacher struct in the generate\_workout function from Listing 13-6:

Filename: src/main.rs

fn generate\_workout(intensity: i32, random\_number: i32) {

let mut expensive\_result = Cacher::new(|num| {

println!("calculating slowly...");

thread::sleep(Duration::from\_secs(2));

num

});

if intensity < 25 {

println!(

"Today, do {} pushups!",

expensive\_result.value(intensity)

);

println!(

"Next, do {} situps!",

expensive\_result.value(intensity)

);

} else {

if random\_number == 3 {

println!("Take a break today! Remember to stay hydrated!");

} else {

println!(

"Today, run for {} minutes!",

expensive\_result.value(intensity)

)

}

}

}

Listing 13-11: Using Cacher in the generate\_workout function to abstract away the caching logic

Instead of saving the closure in a variable directly, we save a new instance of Cacher that holds the closure. Then, in each place we want the result, we call the value method on the Cacher instance. We can call the value method as many times as we want, or not call it at all, and the expensive calculation will be run a maximum of once.

Try running this program with the main function from Listing 13-2, and change the values in the simulated\_user\_specified\_value and simulated\_random\_number variables. Then verify that in all of the cases in the various if and else blocks, calculating slowly... only shows up once and only when needed. The Cacher takes care of the logic necessary to ensure we aren’t calling the expensive calculation more than we need to, so that generate\_workout can focus on the business logic.

Limitations of the Cacher Implementation

Caching values is a generally useful behavior that we might want to use in other parts of our code with different closures. However, there are a few problems with the current implementation of Cacher that would make reusing it in different contexts difficult.

The first problem is a Cacher instance assumes it will always get the same value for the parameter arg to the value method. That is, this test of Cacher will fail:

#[test]

fn call\_with\_different\_values() {

let mut c = Cacher::new(|a| a);

let v1 = c.value(1);

let v2 = c.value(2);

assert\_eq!(v2, 2);

}

This test creates a new Cacher instance with a closure that returns the value passed into it. We call the value method on this Cacher instance with an arg value of 1 and then an arg value of 2, and we expect that the call to value with the arg value of 2 should return 2.

Run this with the Cacher implementation from Listing 13-9 and Listing 13-10 and the test will fail on the assert\_eq! with this message:

thread 'call\_with\_different\_arg\_values' panicked at 'assertion failed:

`(left == right)` (left: `1`, right: `2`)', src/main.rs

The problem is that the first time we called c.value with 1, the Cacher instance saved Some(1) in self.value. After that, no matter what we pass in to the value method, it will always return 1.

Try modifying Cacher to hold a hash map rather than a single value. The keys of the hash map will be the arg values that are passed in, and the values of the hash map will be the result of calling the closure on that key. Instead of looking at whether self.value directly has a Some or a None value, the value function will look up the arg in the hash map and return the value, if it’s present. If it’s not present, the Cacher will call the closure and save the resulting value in the hash map associated with its arg value.

Another problem with the current Cacher implementation is that it only accepts closures that take one parameter of type i32 and return an i32. We might want to cache the results of closures that take a string slice and return usize values, for example. To fix this issue, try introducing more generic parameters to increase the flexibility of the Cacher functionality.

Closures Can Capture Their Environment

In the workout generator example, we only used closures as inline anonymous functions. Closures have an additional ability that functions don’t have, however: they can capture their environment and access variables from the scope in which they’re defined.

Listing 13-12 has an example of a closure stored in the variable equal\_to\_x that uses the variable x from the closure’s surrounding environment:

Filename: src/main.rs

fn main() {

let x = 4;

let equal\_to\_x = |z| z == x;

let y = 4;

assert!(equal\_to\_x(y));

}

Listing 13-12: Example of a closure that refers to a variable in its enclosing scope

Here, even though x is not one of the parameters of equal\_to\_x, the equal\_to\_x closure is allowed to use the x variable that’s defined in the same scope that equal\_to\_x is defined in.

We can’t do the same with functions; let’s see what happens if we try:

Filename: src/main.rs

fn main() {

let x = 4;

fn equal\_to\_x(z: i32) -> bool { z == x }

let y = 4;

assert!(equal\_to\_x(y));

}

We get an error:

error[E0434]: can't capture dynamic environment in a fn item; use the || { ... }

closure form instead

-->

|

4 | fn equal\_to\_x(z: i32) -> bool { z == x }

| ^

The compiler even reminds us that this only works with closures!

When a closure captures a value from its environment, it uses memory to store the values for use in the closure body. This use of memory is overhead that we don’t want to pay in more common cases, where we want to execute code that doesn’t capture its environment. Because functions are never allowed to capture their environment, defining and using functions will never incur this overhead.

Closures can capture values from their environment in three ways, which directly map to the three ways a function can take a parameter: taking ownership, borrowing immutably, and borrowing mutably. These are encoded in the three Fn traits as follows:

FnOnce consumes the variables it captures from its enclosing scope, known as the closure’s environment. In order to consume the captured variables, the closure must take ownership of these variables and move them into the closure when it is defined. The Once part of the name is because the closure can’t take ownership of the same variables more than once, so it can only be called one time.

Fn borrows values from the environment immutably.

FnMut can change the environment since it mutably borrows values.

When we create a closure, Rust infers which to use based on how the closure uses the values from the environment. In Listing 13-12, the equal\_to\_x closure borrows x immutably (so equal\_to\_x has the Fn trait) since the body of the closure only needs to read the value in x.

If we want to force the closure to take ownership of the values it uses in the environment, we can use the move keyword before the parameter list. This is mostly useful when passing a closure to a new thread in order to move the data so that it’s owned by the new thread.

We’ll have more examples of move closures in Chapter 16 when we talk about concurrency, but for now here’s the code from Listing 13-12 with the move keyword added to the closure definition and using vectors instead of integers, since integers can be copied rather than moved:

Filename: src/main.rs

fn main() {

let x = vec![1, 2, 3];

let equal\_to\_x = move |z| z == x;

println!("can't use x here: {:?}", x);

let y = vec![1, 2, 3];

assert!(equal\_to\_x(y));

}

This example doesn’t compile:

error[E0382]: use of moved value: `x`

--> src/main.rs:6:40

|

4 | let equal\_to\_x = move |z| z == x;

| -------- value moved (into closure) here

5 |

6 | println!("can't use x here: {:?}", x);

| ^ value used here after move

|

= note: move occurs because `x` has type `std::vec::Vec<i32>`, which does not

implement the `Copy` trait

The x value is moved into the closure when the closure is defined, because we added the move keyword. The closure then has ownership of x, and main isn’t allowed to use x anymore in the println! statement. Removing println! will fix this example.

Most of the time when specifying one of the Fn trait bounds, you can start with Fn and the compiler will tell you if you need FnMut or FnOnce based on what happens in the closure body.

To illustrate situations where closures that can capture their environment are useful as function parameters, let’s move on to our next topic: iterators.

Processing a Series of Items with Iterators

The iterator pattern allows you to perform some task on a sequence of items in turn. An iterator is responsible for the logic of iterating over each item and determining when the sequence has finished. When we use iterators, we don’t have to reimplement that logic ourselves.

In Rust, iterators are lazy, meaning they have no effect until we call methods on them, which consume the iterator to use it up. For example, the code in Listing 13-13 creates an iterator over the items in the vector v1 by calling the iter method defined on Vec. This code by itself doesn’t do anything useful:

let v1 = vec![1, 2, 3];

let v1\_iter = v1.iter();

Listing 13-13: Creating an iterator

Once we’ve created an iterator, we can choose to use it in a variety of ways. In Listing 3-6 from Chapter 3, we actually used iterators with for loops to execute some code on each item, though we glossed over what the call to iter did until now.

The example in Listing 13-14 separates the creation of the iterator from the use of the iterator in the for loop. The iterator is stored in the v1\_iter variable, and no iteration takes place at that time. Once the for loop is called using the iterator in v1\_iter, then each element in the iterator is used in one iteration of the loop, which prints out each value:

let v1 = vec![1, 2, 3];

let v1\_iter = v1.iter();

for val in v1\_iter {

println!("Got: {}", val);

}

Listing 13-14: Making use of an iterator in a for loop

In languages that don’t have iterators provided by their standard libraries, we would likely write this same functionality by starting a variable at index 0, using that variable to index into the vector to get a value, and incrementing the variable value in a loop until it gets to the total number of items in the vector.

Iterators take care of all of that logic for us, cutting down on repetitive code we could potentially mess up. Iterators give us more flexibility to use the same logic with many different kinds of sequences, not just data structures we can index into like vectors. Let’s see how iterators do that.

The Iterator trait and the next method

Iterators all implement a trait named Iterator that is defined in the standard library. The definition of the trait looks like this:

trait Iterator {

type Item;

fn next(&mut self) -> Option<Self::Item>;

// methods with default implementations elided

}

You’ll notice some new syntax that we haven’t covered yet: type Item and Self::Item, which are defining an associated type with this trait. We’ll talk about associated types in depth in Chapter 19, but for now, all you need to know is that this code says implementing the Iterator trait requires that you also define an Item type, and this Item type is used in the return type of the next method. In other words, the Item type will be the type returned from the iterator.

The Iterator trait only requires implementers to define one method: the next method, which returns one item of the iterator at a time wrapped in Some and, when iteration is over, it returns None.

We can call the next method on iterators directly; Listing 13-15 demonstrates what values are returned from repeated calls to next on the iterator created from the vector:

Filename: src/lib.rs

#[test]

fn iterator\_demonstration() {

let v1 = vec![1, 2, 3];

let mut v1\_iter = v1.iter();

assert\_eq!(v1\_iter.next(), Some(&1));

assert\_eq!(v1\_iter.next(), Some(&2));

assert\_eq!(v1\_iter.next(), Some(&3));

assert\_eq!(v1\_iter.next(), None);

}

Listing 13-15: Calling the next method on an iterator

Note that we needed to make v1\_iter mutable: calling the next method on an iterator changes state that keeps track of where it is in the sequence. Put another way, this code consumes, or uses up, the iterator. Each call to next eats up an item from the iterator. We didn’t need to make v1\_iter mutable when we used a for loop because the loop took ownership of v1\_iter and made it mutable behind the scenes.

Also note that the values we get from the calls to next are immutable references to the values in the vector. The iter method produces an iterator over immutable references. If we want to create an iterator that takes ownership of v1 and returns owned values, we can call into\_iter instead of iter. Similarly, if we want to iterate over mutable references, we can call iter\_mut instead of iter.

Methods in the Iterator Trait that Consume the Iterator

The Iterator trait has a number of different methods with default implementations provided for us by the standard library; you can find out all about these methods by looking in the standard library API documentation for the Iterator trait. Some of these methods call the next method in their definition, which is why we’re required to implement the next method when implementing the Iterator trait.

Methods that call next are called consuming adaptors, because calling them uses up the iterator. One example is the sum method, which takes ownership of the iterator and iterates through the items by repeatedly calling next, thus consuming the iterator. As it iterates through, it adds each item to a running total and returns the total when iteration is complete. Listing 13-16 has a test illustrating a use of the sum method:

Filename: src/lib.rs

#[test]

fn iterator\_sum() {

let v1 = vec![1, 2, 3];

let v1\_iter = v1.iter();

let total: i32 = v1\_iter.sum();

assert\_eq!(total, 6);

}

Listing 13-16: Calling the sum method to get the total of all items in the iterator

We aren’t allowed to use v1\_iter after the call to sum since sum takes ownership of the iterator we call it on.

Methods in the Iterator Trait that Produce Other Iterators

Other methods defined on the Iterator trait, known as iterator adaptors, allow us to change iterators into different kind of iterators. We can chain multiple calls to iterator adaptors to perform complex actions in a readable way. Because all iterators are lazy, however, we have to call one of the consuming adaptor methods in order to get results from calls to iterator adaptors.

Listing 13-17 shows an example of calling the iterator adaptor method map which takes a closure to call on each item in order to produce a new iterator. The closure here creates a new iteratorin which each item from the vector has been incremented by 1. This code produces a warning, though:

Filename: src/main.rs

let v1: Vec<i32> = vec![1, 2, 3];

v1.iter().map(|x| x + 1);

Listing 13-17: Calling the iterator adapter map to create a new iterator

The warning we get is:

warning: unused result which must be used: iterator adaptors are lazy and do

nothing unless consumed

--> src/main.rs:4:1

|

4 | v1.iter().map(|x| x + 1);

| ^^^^^^^^^^^^^^^^^^^^^^^^^

|

= note: #[warn(unused\_must\_use)] on by default

The code in Listing 13-17 isn’t actually doing anything; the closure we’ve specified never gets called. The warning reminds us why: iterator adaptors are lazy, and we need to consume the iterator here.

To fix this and consume the iterator, we’re going to use the collect method, which we saw briefly in Chapter 12. This method consumes the iterator and collects the resulting values into a collection data type.

In Listing 13-18, we collect the results of iterating over the iterator that’s returned from the call to map into a vector. This vector will end up containing each item from the original vector incremented by 1:

Filename: src/main.rs

let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<\_> = v1.iter().map(|x| x + 1).collect();

assert\_eq!(v2, vec![2, 3, 4]);

Listing 13-18: Calling the map method to create a new iterator, then calling the collect method to consume the new iterator and create a vector

Because map takes a closure, we can specify any operation we want to perform on each item. This is a great example of how closures let us customize some behavior while reusing the iteration behavior that the Iterator trait provides.

Using Closures that Capture their Environment with Iterators

Now that we’ve introduced iterators, we can demonstrate a common use of closures that capture their environment by using the filter iterator adapter. The filter method on an iterator takes a closure that takes each item from the iterator and returns a boolean. If the closure returns true, the value will be included in the iterator produced by filter. If the closure returns false, the value won’t be included in the resulting iterator.

In Listing 13-19 we use filter with a closure that captures the shoe\_size variable from its environment, in order to iterate over a collection of Shoe struct instances. It will return only shoes that are the specified size:

Filename: src/lib.rs

#[derive(PartialEq, Debug)]

struct Shoe {

size: i32,

style: String,

}

fn shoes\_in\_my\_size(shoes: Vec<Shoe>, shoe\_size: i32) -> Vec<Shoe> {

shoes.into\_iter()

.filter(|s| s.size == shoe\_size)

.collect()

}

#[test]

fn filters\_by\_size() {

let shoes = vec![

Shoe { size: 10, style: String::from("sneaker") },

Shoe { size: 13, style: String::from("sandal") },

Shoe { size: 10, style: String::from("boot") },

];

let in\_my\_size = shoes\_in\_my\_size(shoes, 10);

assert\_eq!(

in\_my\_size,

vec![

Shoe { size: 10, style: String::from("sneaker") },

Shoe { size: 10, style: String::from("boot") },

]

);

}

Listing 13-19: Using the filter method with a closure that captures shoe\_size

The shoes\_in\_my\_size function takes ownership of a vector of shoes and a shoe size as parameters. It returns a vector containing only shoes of the specified size.

In the body of shoes\_in\_my\_size, we call into\_iter to create an iterator that takes ownership of the vector. Then we call filter to adapt that iterator into a new iterator that only contains elements for which the closure returns true.

The closure captures the shoe\_size parameter from the environment and compares the value with each shoe’s size, keeping only shoes of the size specified. Finally, calling collect gathers the values returned by the adapted iterator into a vector that’s returned by the function.

The test shows that when we call shoes\_in\_my\_size, we only get back shoes that have the same size as the value we specified.

Implementing the Iterator Trait to Create Our Own Iterators

We’ve shown that we can create an iterator by calling iter, into\_iter, or iter\_mut on a vector. We can create iterators from the other collection types in the standard library, such as hash map. We can also create iterators that do anything we want by implementing the Iterator trait on our own types. As previously mentioned, the only method we’re required to provide a definition for is the next method. Once we’ve done that, we can use all other methods that have default implementations provided by the Iterator trait!

To demonstrate, let’s create an iterator that will only ever count from 1 to 5. First, we’ll create a struct to hold to some values, and then we’ll make this struct into an iterator by implementing the Iterator trait and use the values in that implementation.

Listing 13-20 has the definition of the Counter struct and an associated new function to create instances of Counter:

Filename: src/lib.rs

struct Counter {

count: u32,

}

impl Counter {

fn new() -> Counter {

Counter { count: 0 }

}

}

Listing 13-20: Defining the Counter struct and a new function that creates instances of Counter with an initial value of 0 for count

The Counter struct has one field named count. This holds a u32 value that will keep track of where we are in the process of iterating from 1 to 5. The count field is private since we want the implementation of Counter to manage its value. The new function enforces the behavior of always starting new instances with a value of 0 in the count field.

Next, we’re going to implement the Iterator trait for our Counter type by defining the body of the next method, to specify what we want to happen when this iterator is used, as shown in Listing 13-21:

Filename: src/lib.rs

impl Iterator for Counter {

type Item = u32;

fn next(&mut self) -> Option<Self::Item> {

self.count += 1;

if self.count < 6 {

Some(self.count)

} else {

None

}

}

}

Listing 13-21: Implementing the Iterator trait on our Counter struct

We set the associated Item type for our iterator to u32, meaning the iterator will return u32 values. Again, don’t worry about associated types yet, we’ll be covering them in Chapter 19.

We want our iterator to add one to the current state, so we initialized count to 0 so it would return one first. If the value of count is less than six, next will return the current value wrapped in Some, but if count is six or higher, our iterator will return None.

Using Our Counter Iterator’s next Method

Once we’ve implemented the Iterator trait, we have an iterator! Listing 13-22 shows a test demonstrating that we can use the iterator functionality of our Counter struct by calling the next method on it directly, just like we did with the iterator created from a vector in Listing 13-15:

Filename: src/lib.rs

#[test]

fn calling\_next\_directly() {

let mut counter = Counter::new();

assert\_eq!(counter.next(), Some(1));

assert\_eq!(counter.next(), Some(2));

assert\_eq!(counter.next(), Some(3));

assert\_eq!(counter.next(), Some(4));

assert\_eq!(counter.next(), Some(5));

assert\_eq!(counter.next(), None);

}

Listing 13-22: Testing the functionality of the next method implementation

This test creates a new Counter instance in the counter variable and then calls next repeatedly, verifying that we have implemented the behavior we want this iterator to have: returning the values from 1 to 5.

Using Other Iterator Trait Methods on Our Iterator

Because we implemented the Iterator trait by defining the next method, we can now use any Iterator trait method’s default implementations as defined in the standard library, since they all use the next method’s functionality.

For example, if for some reason we wanted to take the values produced by an instance of Counter, pair them with values produced by another Counter instance after skipping the first value, multiply each pair together, keep only those results that are divisible by three, and add all the resulting values together, we could do so as shown in the test in Listing 13-23:

Filename: src/lib.rs

#[test]

fn using\_other\_iterator\_trait\_methods() {

let sum: u32 = Counter::new().zip(Counter::new().skip(1))

.map(|(a, b)| a \* b)

.filter(|x| x % 3 == 0)

.sum();

assert\_eq!(18, sum);

}

Listing 13-23: Using a variety of Iterator trait methods on our Counter iterator

Note that zip produces only four pairs; the theoretical fifth pair (5, None) is never produced because zip returns None when either of its input iterators return None.

All of these method calls are possible because we specified how the next method works, and the standard library provides default implementations for other methods that call next.

Improving our I/O Project

With this new knowledge, we can improve the I/O project in Chapter 12 by using iterators to make places in the code clearer and more concise. Let’s take a look at how iterators can improve our implementation of both the Config::new function and the search function.

Removing a clone Using an Iterator

In Listing 12-6, we added code that took a slice of String values and created an instance of the Config struct by indexing into the slice and cloning the values, allowing the Config struct to own those values. We’ve reproduced the implementation of the Config::new function as it was at the end of Chapter 12 in Listing 13-24:

Filename: src/lib.rs

impl Config {

pub fn new(args: &[String]) -> Result<Config, &'static str> {

if args.len() < 3 {

return Err("not enough arguments");

}

let query = args[1].clone();

let filename = args[2].clone();

let case\_sensitive = env::var("CASE\_INSENSITIVE").is\_err();

Ok(Config { query, filename, case\_sensitive })

}

}

Listing 13-24: Reproduction of the Config::new function from the end of Chapter 12

At the time, we said not to worry about the inefficient clone calls here because we would remove them in the future. Well, that time is now!

We needed clone here because we have a slice with String elements in the parameter args, but the new function doesn’t own args. In order to be able to return ownership of a Config instance, we had to clone the values from the query and filename fields of Config, so that the Config instance can own its values.

With our new knowledge about iterators, we can change the new function to take ownership of an iterator as its argument instead of borrowing a slice. We’ll use the iterator functionality instead of the code that checks the length of the slice and indexes into specific locations. This will clear up what the Config::new function is doing since the iterator will take care of accessing the values.

Once Config::new takes ownership of the iterator and stops using indexing operations that borrow, we can move the String values from the iterator into Config rather than calling clone and making a new allocation.

Using the Iterator Returned by env::args Directly

Open your I/O project’s src/main.rs, and we’ll change the start of the main function that we had at the end of Chapter 12:

Filename: src/main.rs

fn main() {

let args: Vec<String> = env::args().collect();

let config = Config::new(&args).unwrap\_or\_else(|err| {

eprintln!("Problem parsing arguments: {}", err);

process::exit(1);

});

// ...snip...

}

To the code in Listing 13-25:

Filename: src/main.rs

fn main() {

let config = Config::new(env::args()).unwrap\_or\_else(|err| {

eprintln!("Problem parsing arguments: {}", err);

process::exit(1);

});

// ...snip...

}

Listing 13-25: Passing the return value of env::args to Config::new

The env::args function returns an iterator! Rather than collecting the iterator values into a vector and then passing a slice to Config::new, now we’re passing ownership of the iterator returned from env::args to Config::new directly.

Next, we need to update the definition of Config::new. In your I/O project’s src/lib.rs, let’s change the signature of Config::new to look like Listing 13-26:

Filename: src/lib.rs

impl Config {

pub fn new(args: std::env::Args) -> Result<Config, &'static str> {

// ...snip...

Listing 13-26: Updating the signature of Config::new to expect an iterator

The standard library documentation for the env::args function shows that the type of the iterator it returns is std::env::Args. We’ve updated the signature of the Config::new function so that the parameter args has the type std::env::Args instead of &[String].

Using Iterator Trait Methods Instead of Indexing

Next, we’ll fix the body of Config::new. The standard library documentation also mentions that std::env::Args implements the Iterator trait, so we know we can call the next method on it! Listing 13-27 has updated the code from Listing 12-23 to use the next method:

Filename: src/lib.rs

impl Config {

pub fn new(mut args: std::env::Args) -> Result<Config, &'static str> {

args.next();

let query = match args.next() {

Some(arg) => arg,

None => return Err("Didn't get a query string"),

};

let filename = match args.next() {

Some(arg) => arg,

None => return Err("Didn't get a file name"),

};

let case\_sensitive = env::var("CASE\_INSENSITIVE").is\_err();

Ok(Config {

query, filename, case\_sensitive

})

}

}

Listing 13-27: Changing the body of Config::new to use iterator methods

Remember that the first value in the return value of env::args is the name of the program. We want to ignore that and get to the next value, so first we call next and do nothing with the return value. Second, we call next on the value we want to put in the query field of Config. If next returns a Some, we use a match to extract the value. If it returns None, it means not enough arguments were given and we return early with an Err value. We do the same thing for the filename value.

Making Code Clearer with Iterator Adaptors

The other place in our I/O project we could take advantage of iterators is in the search function, reproduced here in Listing 13-28 as it was at the end of Chapter 12:

Filename: src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

let mut results = Vec::new();

for line in contents.lines() {

if line.contains(query) {

results.push(line);

}

}

results

}

Listing 13-28: The implementation of the search function from Chapter 12

We can write this code in a much more concise way using iterator adaptor methods. This also lets us avoid having a mutable intermediate results vector. The functional programming style prefers to minimize the amount of mutable state to make code clearer. Removing the mutable state might make it easier for us to make a future enhancement to make searching happen in parallel, since we wouldn’t have to manage concurrent access to the results vector. Listing 13-29 shows this change:

Filename: src/lib.rs

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {

contents.lines()

.filter(|line| line.contains(query))

.collect()

}

Listing 13-29: Using iterator adaptor methods in the implementation of the search function

Recall that the purpose of the search function is to return all lines in contents that contain the query. Similar to the filter example in Listing 13-19, we can use the filter adaptor to keep only the lines that line.contains(query) returns true for. We then collect the matching lines up into another vector with collect. Much simpler! Feel free to make the same change to use iterator methods in the search\_case\_insensitive function as well.

The next logical question is which style you should choose in your own code and why: the original implementation in Listing 13-28, or the version using iterators in Listing 13-29. Most Rust programmers prefer to use the iterator style. It’s a bit tougher to get the hang of at first, but once you get a feel for the various iterator adaptors and what they do, iterators can be easier to understand. Instead of fiddling with the various bits of looping and building new vectors, the code focuses on the high-level objective of the loop. This abstracts away some of the commonplace code so that it’s easier to see the concepts that are unique to this code, like the filtering condition each element in the iterator must pass.

But are the two implementations truly equivalent? The intuitive assumption might be that the more low-level loop will be faster. Let’s talk about performance.

Comparing Performance: Loops versus Iterators

To determine which to use, we need to know which version of our search functions is faster: the version with an explicit for loop or the version with iterators.

We ran a benchmark by loading the entire contents of “The Adventures of Sherlock Holmes” by Sir Arthur Conan Doyle into a String and looking for the word “the” in the contents. Here were the results of the benchmark on the version of search using the for loop and the version using iterators:

test bench\_search\_for ... bench: 19,620,300 ns/iter (+/- 915,700)

test bench\_search\_iter ... bench: 19,234,900 ns/iter (+/- 657,200)

The iterator version ended up slightly faster! We’re not going to go through the benchmark code here, as the point is not to prove that they’re exactly equivalent, but to get a general sense of how these two implementations compare performance-wise.

For a more comprehensive benchmark, you’d want to check various texts of various sizes, different words, words of different lengths, and all kinds of other variations. The point is this: iterators, while a high-level abstraction, get compiled down to roughly the same code as if you’d written the lower-level code yourself. Iterators are one of Rust’s zero-cost abstractions, by which we mean using the abstraction imposes no additional runtime overhead, in the same way that Bjarne Stroustrup, the original designer and implementer of C++, defines zero-overhead:

In general, C++ implementations obey the zero-overhead principle: What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.

Bjarne Stroustrup “Foundations of C++”

As another example, here is some code taken from an audio decoder. The decoding algorithm uses the linear prediction mathematical operation to estimate future values based on a linear function of the previous samples.

This code uses an iterator chain to do some math on three variables in scope: a buffer slice of data, an array of 12 coefficients, and an amount by which to shift data in qlp\_shift. We’ve declared the variables within this example but not given them any values; while this code doesn’t have much meaning outside of its context, it’s still a concise, real-world example of how Rust translates high-level ideas to low-level code:

let buffer: &mut [i32];

let coefficients: [i64; 12];

let qlp\_shift: i16;

for i in 12..buffer.len() {

let prediction = coefficients.iter()

.zip(&buffer[i - 12..i])

.map(|(&c, &s)| c \* s as i64)

.sum::<i64>() >> qlp\_shift;

let delta = buffer[i];

buffer[i] = prediction as i32 + delta;

}

In order to calculate the value of prediction, this code iterates through each of the 12 values in coefficients and uses the zip method to pair the coefficient values with the previous 12 values in buffer. Then, for each pair, we multiply the values together, sum all the results, and shift the bits in the sum qlp\_shift bits to the right.

Calculations in applications like audio decoders often prioritize performance most highly. Here, we’re creating an iterator, using two adaptors, then consuming the value. What assembly code would this Rust code compile to? Well, as of this writing, it compiles down to the same assembly you’d write by hand. There’s no loop at all corresponding to the iteration over the values in coefficients: Rust knows that there are twelve iterations, so it “unrolls” the loop. Unrolling is an optimization that removes the overhead of the loop controlling code and instead generates repetitive code for each iteration of the loop.

All of the coefficients get stored in registers, which means it’s very fast to access the values. There are no bounds checks on the array access at runtime. All these optimizations Rust is able to apply make the resulting code extremely efficient.

Now that you know this, go use iterators and closures without fear! They make code feel higher-level, but don’t impose a runtime performance penalty for doing so.

Summary

Closures and iterators are Rust features inspired by functional programming language ideas. They contribute to Rust’s ability to clearly express high-level ideas, at low level performance. The implementations of closures and iterators are such that runtime performance is not affected. This is part of Rust’s goal to strive to provide zero-cost abstractions.

Now that we’ve improved the expressiveness of our I/O project, let’s look at some more features of cargo that would help us get ready to share the project with the world.