Functional Language Features: Iterators and Closures

Rust's design has taken inspiration from many 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.

In this chapter, we won't debate the issue of what functional programming is or isn't but will instead discuss some features of Rust that are similar to features in many languages often referred to as functional.

More specifically, we'll cover:

- *Closures*, a function-like construct you can store in a variable
- *Iterators*, a way of processing a series of elements
- How to use closures and iterators to improve the I/O project in Chapter 12
- The performance of closures and iterators (Spoiler alert: they're faster than you might think!)

We've already covered some other Rust features, such as pattern matching and enums, that are also influenced by the functional style. Because mastering closures and iterators is an important part of writing idiomatic, fast Rust code, we'll devote this entire chapter to them.

Closures: Anonymous Functions that 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 elsewhere to evaluate it in a different context. Unlike functions, closures can capture values from the scope in which they're defined. We'll demonstrate how these closure features allow for code reuse and behavior customization.

Capturing the Environment with Closures

We'll first examine how we can use closures to capture values from the environment they're defined in for later use. Here's the scenario: Every so often, our t-shirt company gives away an exclusive, limited-edition shirt to someone on our mailing list as a promotion. People on the mailing list can optionally add their favorite color to their profile. If the person chosen for a free shirt has their favorite color set, they get that color shirt. If the person hasn't specified a favorite color, they get whatever color the company currently has the most of.

There are many ways to implement this. For this example, we're going to use an enum called `ShirtColor` that has the variants `Red` and `Blue` (limiting the number of colors available for simplicity). We represent the company's inventory with an `Inventory` struct that has a field named `shirts` that contains a `Vec<ShirtColor>` representing the shirt colors currently in stock. The method `giveaway` defined on `Inventory` gets the optional shirt color preference of the free shirt winner, and returns the shirt color the person will get. This setup is shown in Listing 13-1:

Filename: src/main.rs

```rust
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}

fn main() {
    let store = Inventory {
        shirts: vec![ShirtColor::Blue, ShirtColor::Red, ShirtColor::Blue],
    };

    let user_pref1 = Some(ShirtColor::Red);
    let giveaway1 = store.giveaway(user_pref1);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref1, giveaway1
    );

    let user_pref2 = None;
    let giveaway2 = store.giveaway(user_pref2);
    println!(
        "The user with preference {:?} gets {:?}",
        user_pref2, giveaway2
    );
}
```

Listing 13-1: Shirt company giveaway situation

The `store` defined in `main` has two blue shirts and one red shirt remaining to distribute for this limited-edition promotion. We call the `giveaway` method for a user with a preference for a red shirt and a user without any preference.

Again, this code could be implemented in many ways, and here, to focus on closures, we've stuck to concepts you've already learned except for the body of the `giveaway` method that uses a closure. In the `giveaway` method, we get the user preference as a parameter of type `Option<ShirtColor>` and call the `unwrap_or_else` method on `user_preference`. The `unwrap_or_else` method on `Option<T>` is defined by the standard library. It takes one argument: a closure without any arguments that returns a value `T` (the same type stored in the `Some` variant of the `Option<T>`, in this case `ShirtColor`). If the `Option<T>` is the `Some` variant, `unwrap_or_else` returns the value from within the `Some`. If the `Option<T>` is the `None` variant, `unwrap_or_else` calls the closure and returns the value returned by the closure.

We specify the closure expression `|| self.most_stocked()` as the argument to `unwrap_or_else`. This is a closure that takes no parameters itself (if the closure had parameters, they would appear between the two vertical bars). The body of the closure calls `self.most_stocked()`. We're defining the closure here, and the implementation of `unwrap_or_else` will evaluate the closure later if the result is needed.

Running this code prints:

```
$ cargo run
   Compiling shirt-company v0.1.0 (file:///projects/shirt-company)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/shirt-company`
The user with preference Some(Red) gets Red
The user with preference None gets Blue
```

One interesting aspect here is that we've passed a closure that calls `self.most_stocked()` on the current `Inventory` instance. The standard library didn't need to know anything about the `Inventory` or `ShirtColor` types we defined, or the logic we want to use in this scenario. The closure captures an immutable reference to the `self Inventory` instance and passes it with the code we specify to the `unwrap_or_else` method. Functions, on the other hand, are not able to capture their environment in this way.

Closure Type Inference and Annotation

There are more differences between functions and closures. Closures don't usually require you to annotate the types of the parameters or the return value like `fn` functions do. Type

annotations are required on functions because the types 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, on the other hand, aren't used in an exposed interface like this: they're stored in variables and used without naming them and exposing them to users of our library.

Closures are typically short and relevant only within a narrow context rather than in any arbitrary scenario. Within these limited contexts, the compiler can infer the types of the parameters and the return type, similar to how it's able to infer the types of most variables (there are rare cases where the compiler needs closure type annotations too).

As with variables, we can 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 a closure would look like the definition shown in Listing 13-2. In this example, we're defining a closure and storing it in a variable rather than defining the closure in the spot we pass it as an argument as we did in Listing 13-1.

Filename: src/main.rs

```
let expensive_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num
};
```

Listing 13-2: Adding optional type annotations of the parameter and return value types in the closure

With type annotations added, the syntax of closures looks more similar to the syntax of functions. Here we define a function that adds 1 to its parameter and a closure that has the same behavior, for comparison. We've added some spaces 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: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

IMP

The first line shows a function definition, and the second line shows a fully annotated closure definition. In the third line, we remove the type annotations from the closure definition. In the fourth line, we remove the brackets, which are optional because the closure body has only one expression. These are all valid definitions that will produce the same behavior when they're called. The `add_one_v3` and `add_one_v4` lines require the closures to be evaluated to be able to compile because the types will be inferred from their usage. This is similar to `let v =`

`Vec::new();` needing either type annotations or values of some type to be inserted into the `Vec` for Rust to be able to infer the type.

For closure definitions, the compiler will infer one concrete type for each of their parameters and for their return value. For instance, Listing 13-3 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. Because there are no type annotations, we can call the closure with any type, which we've done here with `String` the first time. If we then try to call `example_closure` with an integer, 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 13-3: Attempting to call a closure whose types are inferred with two different types

The compiler gives us this error:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |             --------------- ^- help: try using a conversion method:
`.to_string()`
  |             |               |
  |             |               expected `String`, found integer
  |             arguments to this function are incorrect
  |
note: expected because the closure was earlier called with an argument of type
`String`
 --> src/main.rs:4:29
  |
4 |     let s = example_closure(String::from("hello"));
  |             --------------- ^^^^^^^^^^^^^^^^^^^^ expected because this
argument is of type `String`
  |             |
  |             in this closure call
note: closure parameter defined here
 --> src/main.rs:2:28
  |
2 |     let example_closure = |x| x;
  |                            ^

For more information about this error, try `rustc --explain E0308`.
error: could not compile `closure-example` (bin "closure-example") due to 1
previous error
```

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 into the closure in `example_closure`, and we get a type error when we next try to use a different type with the same closure.

IMP

Capturing References or Moving Ownership

IMP

Closures can capture values from their environment in three ways, which directly map to the three ways a function can take a parameter: borrowing immutably, borrowing mutably, and taking ownership. The closure will decide which of these to use based on what the body of the function does with the captured values.

In Listing 13-4, we define a closure that captures an immutable reference to the vector named `list` because it only needs an immutable reference to print the value:

Filename: src/main.rs

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let only_borrows = || println!("From closure: {list:?}");

    println!("Before calling closure: {list:?}");
    only_borrows();
    println!("After calling closure: {list:?}");
}
```

Listing 13-4: Defining and calling a closure that captures an immutable reference

This example also illustrates that a variable can bind to a closure definition, and we can later call the closure by using the variable name and parentheses as if the variable name were a function name.

Because we can have multiple immutable references to `list` at the same time, `list` is still accessible from the code before the closure definition, after the closure definition but before the closure is called, and after the closure is called. This code compiles, runs, and prints:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

Next, in Listing 13-5, we change the closure body so that it adds an element to the `list` vector. The closure now captures a mutable reference:

Filename: src/main.rs

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {list:?}");
}
```

Listing 13-5: Defining and calling a closure that captures a mutable reference

This code compiles, runs, and prints:

```
$ cargo run
   Compiling closure-example v0.1.0 (file:///projects/closure-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/closure-example`
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

V V IMP

Note that there's no longer a `println!` between the definition and the call of the `borrows_mutably` closure: when `borrows_mutably` is defined, it captures a mutable reference to `list`. We don't use the closure again after the closure is called, so the mutable borrow ends. Between the closure definition and the closure call, an immutable borrow to print isn't allowed because no other borrows are allowed when there's a mutable borrow. Try adding a `println!` there to see what error message you get!

If you want to force the closure to take ownership of the values it uses in the environment even though the body of the closure doesn't strictly need ownership, you can use the `move` keyword before the parameter list.

This technique is mostly useful when passing a closure to a new thread to move the data so that it's owned by the new thread. We'll discuss threads and why you would want to use them in detail in Chapter 16 when we talk about concurrency, but for now, let's briefly explore spawning a new thread using a closure that needs the `move` keyword. Listing 13-6 shows Listing 13-4 modified to print the vector in a new thread rather than in the main thread:

Filename: src/main.rs

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {list:?}");

    thread::spawn(move || println!("From thread: {list:?}"))
        .join()
        .unwrap();
}
```

Listing 13-6: Using `move` to force the closure for the thread to take ownership of `list`

We spawn a new thread, giving the thread a closure to run as an argument. The closure body prints out the list. In Listing 13-4, the closure only captured `list` using an immutable reference because that's the least amount of access to `list` needed to print it. In this example, even though the closure body still only needs an immutable reference, we need to specify that `list` should be moved into the closure by putting the `move` keyword at the beginning of the closure definition. The new thread might finish before the rest of the main thread finishes, or the main

thread might finish first. If the main thread maintained ownership of `list` but ended before the new thread did and dropped `list`, the immutable reference in the thread would be invalid. Therefore, the compiler requires that `list` be moved into the closure given to the new thread so the reference will be valid. Try removing the `move` keyword or using `list` in the main thread after the closure is defined to see what compiler errors you get!

Moving Captured Values Out of Closures and the `Fn` Traits

Once a closure has captured a reference or captured ownership of a value from the environment where the closure is defined (thus affecting what, if anything, is moved *into* the closure), the code in the body of the closure defines what happens to the references or values when the closure is evaluated later (thus affecting what, if anything, is moved *out of* the closure). A closure body can do any of the following: move a captured value out of the closure, mutate the captured value, neither move nor mutate the value, or capture nothing from the environment to begin with.

The way a closure captures and handles values from the environment affects which traits the closure implements, and traits are how functions and structs can specify what kinds of closures they can use. Closures will automatically implement one, two, or all three of these `Fn` traits, in an additive fashion, depending on how the closure's body handles the values:

1. `FnOnce` applies to closures that can be called once. All closures implement at least this trait, because all closures can be called. A closure that moves captured values out of its body will only implement `FnOnce` and none of the other `Fn` traits, because it can only be called once.
2. `FnMut` applies to closures that don't move captured values out of their body, but that might mutate the captured values. These closures can be called more than once.
3. `Fn` applies to closures that don't move captured values out of their body and that don't mutate captured values, as well as closures that capture nothing from their environment. These closures can be called more than once without mutating their environment, which is important in cases such as calling a closure multiple times concurrently.

Let's look at the definition of the `unwrap_or_else` method on `Option<T>` that we used in Listing 13-1:

```
impl<T> Option<T> {
    pub fn unwrap_or_else<F>(self, f: F) -> T
    where
        F: FnOnce() -> T
    {
        match self {
            Some(x) => x,
            None => f(),
        }
    }
}
```

Recall that `T` is the generic type representing the type of the value in the `Some` variant of an `Option`. That type `T` is also the return type of the `unwrap_or_else` function: code that calls `unwrap_or_else` on an `Option<String>`, for example, will get a `String`.

Next, notice that the `unwrap_or_else` function has the additional generic type parameter `F`. The `F` type is the type of the parameter named `f`, which is the closure we provide when calling `unwrap_or_else`.

The trait bound specified on the generic type `F` is `FnOnce() -> T`, which means `F` must be able to be called once, take no arguments, and return a `T`. Using `FnOnce` in the trait bound expresses the constraint that `unwrap_or_else` is only going to call `f` at most one time. In the body of `unwrap_or_else`, we can see that if the `Option` is `Some`, `f` won't be called. If the `Option` is `None`, `f` will be called once. Because all closures implement `FnOnce`, `unwrap_or_else` accepts all three kinds of closures and is as flexible as it can be.

> Note: Functions can implement all three of the `Fn` traits too. If what we want to do doesn't require capturing a value from the environment, we can use the name of a function rather than a closure where we need something that implements one of the `Fn` traits. For example, on an `Option<Vec<T>>` value, we could call `unwrap_or_else(Vec::new)` to get a new, empty vector if the value is `None`.

Now let's look at the standard library method `sort_by_key` defined on slices, to see how that differs from `unwrap_or_else` and why `sort_by_key` uses `FnMut` instead of `FnOnce` for the trait bound. The closure gets one argument in the form of a reference to the current item in the slice being considered, and returns a value of type `K` that can be ordered. This function is useful when you want to sort a slice by a particular attribute of each item. In Listing 13-7, we have a list of `Rectangle` instances and we use `sort_by_key` to order them by their `width` attribute from low to high:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{list:#?}");
}
```

Listing 13-7: Using `sort_by_key` to order rectangles by width

This code prints:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/rectangles`
[
    Rectangle {
        width: 3,
        height: 5,
    },
    Rectangle {
        width: 7,
        height: 12,
    },
    Rectangle {
        width: 10,
        height: 1,
    },
]
```

The reason `sort_by_key` is defined to take an `FnMut` closure is that it calls the closure multiple times: once for each item in the slice. The closure `|r| r.width` doesn't capture, mutate, or move out anything from its environment, so it meets the trait bound requirements.

In contrast, Listing 13-8 shows an example of a closure that implements just the `FnOnce` trait, because it moves a value out of the environment. The compiler won't let us use this closure with `sort_by_key`:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut sort_operations = vec![];
    let value = String::from("closure called");

    list.sort_by_key(|r| {
        sort_operations.push(value);
        r.width
    });
    println!("{list:#?}");
}
```

Listing 13-8: Attempting to use an `FnOnce` closure with `sort_by_key`

This is a contrived, convoluted way (that doesn't work) to try and count the number of times `sort_by_key` calls the closure when sorting `list`. This code attempts to do this counting by pushing `value`—a `String` from the closure's environment—into the `sort_operations` vector. The closure captures `value` then moves `value` out of the closure by transferring ownership of `value` to the `sort_operations` vector. This closure can be called once; trying to call it a second time wouldn't work because `value` would no longer be in the environment to be pushed into `sort_operations` again! Therefore, this closure only implements `FnOnce`. When we try to compile this code, we get this error that `value` can't be moved out of the closure because the closure must implement `FnMut`:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
error[E0507]: cannot move out of `value`, a captured variable in an `FnMut`
closure
  --> src/main.rs:18:30
   |
15 |     let value = String::from("closure called");
   |         ----- captured outer variable
16 |
17 |     list.sort_by_key(|r| {
   |                      --- captured by this `FnMut` closure
18 |         sort_operations.push(value);
   |                              ^^^^^ move occurs because `value` has type
`String`, which does not implement the `Copy` trait

For more information about this error, try `rustc --explain E0507`.
error: could not compile `rectangles` (bin "rectangles") due to 1 previous error
```

The error points to the line in the closure body that moves `value` out of the environment. To fix this, we need to change the closure body so that it doesn't move values out of the environment. To count the number of times the closure is called, keeping a counter in the environment and incrementing its value in the closure body is a more straightforward way to calculate that. The closure in Listing 13-9 works with `sort_by_key` because it is only capturing a mutable reference to the `num_sort_operations` counter and can therefore be called more than once:

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{list:#?}, sorted in {num_sort_operations} operations");
}
```

Listing 13-9: Using an `FnMut` closure with `sort_by_key` is allowed

The `Fn` traits are important when defining or using functions or types that make use of closures. In the next section, we'll discuss iterators. Many iterator methods take closure arguments, so keep these closure details in mind as we continue!

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 you use iterators, you don't have to reimplement that logic yourself.

In Rust, iterators are *lazy*, meaning they have no effect until you call methods that consume the iterator to use it up. For example, the code in Listing 13-10 creates an iterator over the items in the vector `v1` by calling the `iter` method defined on `Vec<T>`. This code by itself doesn't do anything useful.

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

let v1_iter = v1.iter();
```

Listing 13-10: Creating an iterator

The iterator is stored in the `v1_iter` variable. Once we've created an iterator, we can use it in a variety of ways. In Listing 3-5 in Chapter 3, we iterated over an array using a `for` loop to execute some code on each of its items. Under the hood this implicitly created and then consumed an iterator, but we glossed over how exactly that works until now.

In the example in Listing 13-11, we separate the creation of the iterator from the use of the iterator in the `for` loop. When the `for` loop is called using the iterator in `v1_iter`, 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 13-11: Using an iterator in a `for` loop

In languages that don't have iterators provided by their standard libraries, you 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 reached the total number of items in the vector.

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

The `Iterator` Trait and the `next` Method

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

```
pub trait Iterator {
    type Item;

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

    // methods with default implementations elided
}
```

Notice this definition uses some new syntax: `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. 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 implementors 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, returns `None`.

We can call the `next` method on iterators directly; Listing 13-12 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 13-12: Calling the `next` method on an iterator

Note that we needed to make `v1_iter` mutable: calling the `next` method on an iterator changes internal state that the iterator uses to keep track of where it is in the sequence. In other words, 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 that Consume the Iterator

The `Iterator` trait has a number of different methods with default implementations provided by the standard library; you can find out 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 you'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 13-13 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 13-13: 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` because `sum` takes ownership of the iterator we call it on.

Methods that Produce Other Iterators

Iterator adaptors are methods defined on the `Iterator` trait that don't consume the iterator. Instead, they produce different iterators by changing some aspect of the original iterator.

Listing 13-14 shows an example of calling the iterator adaptor method `map`, which takes a closure to call on each item as the items are iterated through. The `map` method returns a new iterator that produces the modified items. The closure here creates a new iterator in which each item from the vector will be incremented by 1:

Filename: src/main.rs

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

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

Listing 13-14: Calling the iterator adaptor `map` to create a new iterator

However, this code produces a warning:

```
$ cargo run
   Compiling iterators v0.1.0 (file:///projects/iterators)
warning: unused `Map` that must be used
 --> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: iterators are lazy and do nothing unless consumed
  = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
  |
4 |     let _ = v1.iter().map(|x| x + 1);
  |     +++++++

warning: `iterators` (bin "iterators") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.47s
     Running `target/debug/iterators`
```

The code in Listing 13-14 doesn't do 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 warning and consume the iterator, we'll use the `collect` method, which we used in Chapter 12 with `env::args` in Listing 12-1. This method consumes the iterator and collects the resulting values into a collection data type.

In Listing 13-15, 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 13-15: Calling the `map` method to create a new iterator and 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 you customize some behavior while reusing the iteration behavior that the `Iterator` trait provides.

You can chain multiple calls to iterator adaptors to perform complex actions in a readable way. But because all iterators are lazy, you have to call one of the consuming adaptor methods to get results from calls to iterator adaptors.

Using Closures that Capture Their Environment

Many iterator adapters take closures as arguments, and commonly the closures we'll specify as arguments to iterator adapters will be closures that capture their environment.

For this example, we'll use the `filter` method that takes a closure. The closure gets an item from the iterator and returns a `bool`. If the closure returns `true`, the value will be included in the iteration produced by `filter`. If the closure returns `false`, the value won't be included.

In Listing 13-16, we use `filter` with a closure that captures the `shoe_size` variable from its environment 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: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[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_size(shoes, 10);

        assert_eq!(
            in_my_size,
            vec![
                Shoe {
                    size: 10,
                    style: String::from("sneaker")
                },
                Shoe {
                    size: 10,
                    style: String::from("boot")
                },
            ]
        );
    }
}
```

Listing 13-16: Using the `filter` method with a closure that captures `shoe_size`

The `shoes_in_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_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_size`, we get back only shoes that have the same size as the value we specified.

Improving Our I/O Project

With this new knowledge about iterators, 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 look at how iterators can improve our implementation of the `Config::build` 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. In Listing 13-17, we've reproduced the implementation of the `Config::build` function as it was in Listing 12-23:

Filename: src/lib.rs

```
impl Config {
    pub fn build(args: &[String]) -> Result<Config, &'static str> {
        if args.len() < 3 {
            return Err("not enough arguments");
        }

        let query = args[1].clone();
        let file_path = args[2].clone();

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

Listing 13-17: Reproduction of the `Config::build` function from Listing 12-23

At the time, we said not to worry about the inefficient `clone` calls 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 `build` function doesn't own `args`. To return ownership of a `Config` instance, we had

to clone the values from the `query` and `file_path` fields of `Config` so the `Config` instance can own its values.

With our new knowledge about iterators, we can change the `build` 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 clarify what the `Config::build` function is doing because the iterator will access the values.

Once `Config::build` 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 Returned Iterator Directly

Open your I/O project's *src/main.rs* file, which should look like this:

Filename: src/main.rs

```
fn main() {
    let args: Vec<String> = env::args().collect();

    let config = Config::build(&args).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

We'll first change the start of the `main` function that we had in Listing 12-24 to the code in Listing 13-18, which this time uses an iterator. This won't compile until we update `Config::build` as well.

Filename: src/main.rs

```
fn main() {
    let config = Config::build(env::args()).unwrap_or_else(|err| {
        eprintln!("Problem parsing arguments: {err}");
        process::exit(1);
    });

    // --snip--
}
```

Listing 13-18: Passing the return value of `env::args` to `Config::build`

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

Next, we need to update the definition of `Config::build`. In your I/O project's *src/lib.rs* file, let's change the signature of `Config::build` to look like Listing 13-19. This still won't compile because we need to update the function body.

Filename: src/lib.rs

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        // --snip--
```

Listing 13-19: Updating the signature of `Config::build` 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`, and that type implements the `Iterator` trait and returns `String` values.

We've updated the signature of the `Config::build` function so the parameter `args` has a generic type with the trait bounds `impl Iterator<Item = String>` instead of `&[String]`. This usage of the `impl Trait` syntax we discussed in the "Traits as Parameters" section of Chapter 10 means that `args` can be any type that implements the `Iterator` trait and returns `String` items.

Because we're taking ownership of `args` and we'll be mutating `args` by iterating over it, we can add the `mut` keyword into the specification of the `args` parameter to make it mutable.

Using `Iterator` Trait Methods Instead of Indexing

Next, we'll fix the body of `Config::build`. Because `args` implements the `Iterator` trait, we know we can call the `next` method on it! Listing 13-20 updates the code from Listing 12-23 to use the `next` method:

Filename: src/lib.rs

```
impl Config {
    pub fn build(
        mut args: impl Iterator<Item = String>,
    ) -> Result<Config, &'static str> {
        args.next();

        let query = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
        };

        let file_path = match args.next() {
            Some(arg) => arg,
            None => return Err("Didn't get a file path"),
        };

        let ignore_case = env::var("IGNORE_CASE").is_ok();

        Ok(Config {
            query,
            file_path,
            ignore_case,
        })
    }
}
```

Listing 13-20: Changing the body of `Config::build` 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` to get 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 `file_path` value.

Making Code Clearer with Iterator Adaptors

We can also take advantage of iterators in the `search` function in our I/O project, which is reproduced here in Listing 13-21 as it was in Listing 12-19:

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 13-21: The implementation of the `search` function from Listing 12-19

We can write this code in a more concise way using iterator adaptor methods. Doing so 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 enable a future enhancement to make searching happen in parallel, because we wouldn't have to manage concurrent access to the `results` vector. Listing 13-22 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 13-22: 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 13-16, this code uses the `filter` adaptor to keep only the lines that `line.contains(query)` returns `true` for. We then collect the matching lines 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.

Choosing Between Loops or Iterators

The next logical question is which style you should choose in your own code and why: the original implementation in Listing 13-21 or the version using iterators in Listing 13-22. 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 it's easier to see the concepts that are unique to this code, such as 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 vs. Iterators

To determine whether to use loops or iterators, you need to know which implementation is faster: the version of the `search` function 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 are 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 was slightly faster! We won't explain the benchmark code here, because the point is not to prove that the two versions are equivalent but to get a general sense of how these two implementations compare performance-wise.

For a more comprehensive benchmark, you should check using various texts of various sizes as the `contents`, different words and words of different lengths as the `query`, and all kinds of other variations. The point is this: iterators, although 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. This is analogous to how Bjarne Stroustrup, the original designer and implementor of C++, defines *zero-overhead* in "Foundations of C++" (2012):

> 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.

As another example, the following code is 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; although 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;
}
```

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, and 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 12 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 accessing the values is very fast. There are no bounds checks on the array access at runtime. All these optimizations that Rust is able to apply make the resulting code extremely efficient. Now that you know this, you can use iterators and closures without fear! They make code seem like it's 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 capability 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 will help us share the project with the world.