Method Syntax

Methods are similar to functions: we declare them with the `fn` keyword and a name, they can have parameters and a return value, and they contain some code that's run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum or a trait object, which we cover in Chapters 6 and 17, respectively), and their first parameter is always `self`, which represents the instance of the struct the method is being called on.

Defining Methods

Let's change the `area` function that has a `Rectangle` instance as a parameter and instead make an `area` method defined on the `Rectangle` struct, as shown in Listing 5-13.

Filename: src/main.rs

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

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-13: Defining an `area` method on the `Rectangle` struct

To define the function within the context of `Rectangle`, we start an `impl` (implementation) block for `Rectangle`. Everything within this `impl` block will be associated with the `Rectangle` type. Then we move the `area` function within the `impl` curly brackets and change the first (and in this case, only) parameter to be `self` in the signature and everywhere within the body. In `main`, where we called the `area` function and passed `rect1` as an argument, we can instead use *method syntax* to call the `area` method on our `Rectangle` instance. The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.

In the signature for `area`, we use `&self` instead of `rectangle: &Rectangle`. The `&self` is actually short for `self: &Self`. Within an `impl` block, the type `Self` is an alias for the type that the `impl` block is for. Methods must have a parameter named `self` of type `Self` for their first parameter, so Rust lets you abbreviate this with only the name `self` in the first parameter spot. Note that we still need to use the `&` in front of the `self` shorthand to indicate this method borrows the `Self` instance, just as we did in `rectangle: &Rectangle`. Methods can take ownership of `self`, borrow `self` immutably as we've done here, or borrow `self` mutably, just as they can any other parameter.

We've chosen `&self` here for the same reason we used `&Rectangle` in the function version: we don't want to take ownership, and we just want to read the data in the struct, not write to it. If we wanted to change the instance that we've called the method on as part of what the method does, we'd use `&mut self` as the first parameter. Having a method that takes ownership of the instance by using just `self` as the first parameter is rare; this technique is usually used when the method transforms `self` into something else and you want to prevent the caller from using the original instance after the transformation.

The main reason for using methods instead of functions, in addition to providing method syntax and not having to repeat the type of `self` in every method's signature, is for organization. We've put all the things we can do with an instance of a type in one `impl` block rather than making future users of our code search for capabilities of `Rectangle` in various places in the library we provide.

Note that we can choose to give a method the same name as one of the struct's fields. For example, we can define a method on `Rectangle` also named `width`:

Filename: src/main.rs

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

Here, we're choosing to make the `width` method return `true` if the value in the instance's `width` field is greater than 0, and `false` if the value is 0: we can use a field within a method of the same name for any purpose. In `main`, when we follow `rect1.width` with parentheses, Rust knows we mean the method `width`. When we don't use parentheses, Rust knows we mean the field `width`.

Often, but not always, when we give methods with the same name as a field we want it to only return the value in the field and do nothing else. Methods like this are called *getters*, and Rust does not implement them automatically for struct fields as some other languages do. Getters are useful because you can make the field private but the method public and thus enable read-only access to that field as part of the type's public API. We will be discussing what public and private are and how to designate a field or method as public or private in Chapter 7.

Where's the `->` Operator?

In C and C++, two different operators are used for calling methods: you use `.` if you're calling a method on the object directly and `->` if you're calling the method on a pointer to the object and need to dereference the pointer first. In other words, if `object` is a pointer, `object->something()` is similar to `(*object).something()`.

Rust doesn't have an equivalent to the `->` operator; instead, Rust has a feature called *automatic referencing and dereferencing*. Calling methods is one of the few places in Rust that has this behavior.

Here's how it works: when you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so `object` matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (`&self`), mutating (`&mut self`), or consuming (`self`). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

Methods with More Parameters

Let's practice using methods by implementing a second method on the `Rectangle` struct. This time, we want an instance of `Rectangle` to take another instance of `Rectangle` and return `true` if the second `Rectangle` can fit completely within `self` (the first `Rectangle`); otherwise it should return

`false`. That is, once we've defined the `can_hold` method, we want to be able to write the program shown in Listing 5-14.

Filename: src/main.rs

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Listing 5-14: Using the as-yet-unwritten `can_hold` method

And the expected output would look like the following, because both dimensions of `rect2` are smaller than the dimensions of `rect1` but `rect3` is wider than `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

We know we want to define a method, so it will be within the `impl Rectangle` block. The method name will be `can_hold`, and it will take an immutable borrow of another `Rectangle` as a parameter. We can tell what the type of the parameter will be by looking at the code that calls the method: `rect1.can_hold(&rect2)` passes in `&rect2`, which is an immutable borrow to `rect2`, an instance

of `Rectangle`. This makes sense because we only need to read `rect2` (rather than write, which would mean we'd need a mutable borrow), and we want `main` to retain ownership of `rect2` so we can use it again after calling the `can_hold` method. The return value of `can_hold` will be a Boolean, and the implementation will check whether the width and height of `self` are both greater than the width and height of the other `Rectangle`, respectively. Let's add the new `can_hold` method to the `impl` block from Listing 5-13, shown in Listing 5-15.

Filename: src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Listing 5-15: Implementing the `can_hold` method on `Rectangle` that takes another `Rectangle` instance as a parameter

When we run this code with the `main` function in Listing 5-14, we'll get our desired output. Methods can take multiple parameters that we add to the signature after the `self` parameter, and those parameters work just like parameters in functions.

Associated Functions

All functions defined within an `impl` block are called *associated functions* because they're associated with the type named after the `impl`. We can define associated functions that don't have `self` as their first parameter (and thus are not methods) because they don't need an instance of the type to

work with. We've already used one function like this: the `String::from` function that's defined on the `String` type.

Associated functions that aren't methods are often used for constructors that will return a new instance of the struct. For example, we could provide an associated function that would have one dimension parameter and use that as both width and height, thus making it easier to create a square `Rectangle` rather than having to specify the same value twice:

Filename: src/main.rs

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}
```

To call this associated function, we use the `::` syntax with the struct name; `let sq = Rectangle::square(3);` is an example. This function is namespaced by the struct: the `::` syntax is used for both associated functions and namespaces created by modules. We'll discuss modules in Chapter 7.

Multiple `impl` Blocks

Each struct is allowed to have multiple `impl` blocks. For example, Listing 5-15 is equivalent to the code shown in Listing 5-16, which has each method in its own `impl` block.

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Listing 5-16: Rewriting Listing 5-15 using multiple `impl` blocks

There's no reason to separate these methods into multiple `impl` blocks here, but this is valid syntax. We'll see a case in which multiple `impl` blocks are useful in Chapter 10, where we discuss generic types and traits.

Summary

Structs let you create custom types that are meaningful for your domain. By using structs, you can keep associated pieces of data connected to each other and name each piece to make your code clear. In `impl` blocks, you can define functions that are associated with your type, and methods are a kind of associated function that let you specify the behavior that instances of your structs have.

But structs aren't the only way you can create custom types: let's turn to Rust's enum feature to add another tool to your toolbox.

Enums and Pattern Matching

In this chapter we'll look at *enumerations*, also referred to as *enums*. Enums allow you to define a type by enumerating its possible *variants*. First, we'll define and use an enum to show how an enum can encode meaning along with data. Next, we'll explore a particularly useful enum, called `Option`, which expresses that a value can be either something or nothing. Then we'll look at how pattern matching in the `match` expression makes it easy to run different code for different values of an enum. Finally, we'll cover how the `if let` construct is another convenient and concise idiom available to handle enums in your code.

Enums are a feature in many languages, but their capabilities differ in each language. Rust's enums are most similar to *algebraic data types* in functional languages, such as F#, OCaml, and Haskell.

Defining an Enum

Enums are a way of defining custom data types in a different way than you do with structs. Let's look at a situation we might want to express in code and see why enums are useful and more appropriate than structs in this case. Say we need to work with IP addresses. Currently, two major standards are used for IP addresses: version four and version six. Because these are the only possibilities for an IP address that our program will come across, we can *enumerate* all possible variants, which is where enumeration gets its name.

Any IP address can be either a version four or a version six address, but not both at the same time. That property of IP addresses makes the enum data structure appropriate, because an enum value can only be one of its variants. Both version four and version six addresses are still fundamentally IP addresses, so they should be treated as the same type when the code is handling situations that apply to any kind of IP address.

We can express this concept in code by defining an `IpAddrKind` enumeration and listing the possible kinds an IP address can be, `V4` and `V6`. These are the variants of the enum:

```
enum IpAddrKind {
    V4,
    V6,
}
```

`IpAddrKind` is now a custom data type that we can use elsewhere in our code.

Enum Values

We can create instances of each of the two variants of `IpAddrKind` like this:

```
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
```

Note that the variants of the enum are namespaced under its identifier, and we use a double colon to separate the two. This is useful because now both values `IpAddrKind::V4` and `IpAddrKind::V6` are of the same type: `IpAddrKind`. We can then, for instance, define a function that takes any `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) {}
```

And we can call this function with either variant:

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

Using enums has even more advantages. Thinking more about our IP address type, at the moment we don't have a way to store the actual IP address *data*; we only know what *kind* it is. Given that you just learned about structs in Chapter 5, you might be tempted to tackle this problem with structs as shown in Listing 6-1.

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Listing 6-1: Storing the data and `IpAddrKind` variant of an IP address using a `struct`

Here, we've defined a struct `IpAddr` that has two fields: a `kind` field that is of type `IpAddrKind` (the enum we defined previously) and an `address` field of type `String`. We have two instances of this struct. The first is `home`, and it has the value `IpAddrKind::V4` as its `kind` with associated address data of `127.0.0.1`. The second instance is `loopback`. It has the other variant of `IpAddrKind` as its `kind` value, `V6`, and has address `::1` associated with it. We've used a struct to bundle the `kind` and `address` values together, so now the variant is associated with the value.

However, representing the same concept using just an enum is more concise: rather than an enum inside a struct, we can put data directly into each enum variant. This new definition of the `IpAddr` enum says that both `V4` and `V6` variants will have associated `String` values:

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

We attach data to each variant of the enum directly, so there is no need for an extra struct. Here it's also easier to see another detail of how enums work: the name of each enum variant that we define also becomes a function that constructs an instance of the enum. That is, `IpAddr::V4()` is a function call that takes a `String` argument and returns an instance of the `IpAddr` type. We automatically get this constructor function defined as a result of defining the enum.

There's another advantage to using an enum rather than a struct: each variant can have different types and amounts of associated data. Version four type IP addresses will always have four numeric components that will have values between 0 and 255. If we wanted to store `V4` addresses as four `u8` values but still express `V6` addresses as one `String` value, we wouldn't be able to with a struct. Enums handle this case with ease:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

We've shown several different ways to define data structures to store version four and version six IP addresses. However, as it turns out, wanting to store IP addresses and encode which kind they are is so common that the standard library has a definition we can use! Let's look at how the standard library defines `IpAddr`: it has the exact enum and variants that we've defined and used, but it

embeds the address data inside the variants in the form of two different structs, which are defined differently for each variant:

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

This code illustrates that you can put any kind of data inside an enum variant: strings, numeric types, or structs, for example. You can even include another enum! Also, standard library types are often not much more complicated than what you might come up with.

Note that even though the standard library contains a definition for `IpAddr`, we can still create and use our own definition without conflict because we haven't brought the standard library's definition into our scope. We'll talk more about bringing types into scope in Chapter 7.

Let's look at another example of an enum in Listing 6-2: this one has a wide variety of types embedded in its variants.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Listing 6-2: A `Message` enum whose variants each store different amounts and types of values

This enum has four variants with different types:

- `Quit` has no data associated with it at all.
- `Move` has named fields like a struct does.
- `Write` includes a single `String`.
- `ChangeColor` includes three `i32` values.

Defining an enum with variants such as the ones in Listing 6-2 is similar to defining different kinds of struct definitions, except the enum doesn't use the `struct` keyword and all the variants are grouped together under the `Message` type. The following structs could hold the same data that the preceding enum variants hold:

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

But if we used the different structs, which each have their own type, we couldn't as easily define a function to take any of these kinds of messages as we could with the `Message` enum defined in Listing 6-2, which is a single type.

There is one more similarity between enums and structs: just as we're able to define methods on structs using `impl`, we're also able to define methods on enums. Here's a method named `call` that we could define on our `Message` enum:

```
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

The body of the method would use `self` to get the value that we called the method on. In this example, we've created a variable `m` that has the value `Message::Write(String::from("hello"))`, and that is what `self` will be in the body of the `call` method when `m.call()` runs.

Let's look at another enum in the standard library that is very common and useful: `Option`.

The `Option` Enum and Its Advantages Over Null Values

This section explores a case study of `Option`, which is another enum defined by the standard library. The `Option` type encodes the very common scenario in which a value could be something or it could be nothing. For example, if you request the first of a list containing items, you would get a value. If you request the first item of an empty list, you would get nothing. Expressing this concept in terms of the type system means the compiler can check whether you've handled all the cases you should be handling; this functionality can prevent bugs that are extremely common in other programming languages.

Programming language design is often thought of in terms of which features you include, but the features you exclude are important too. Rust doesn't have the null feature that many other languages have. *Null* is a value that means there is no value there. In languages with null, variables can always be in one of two states: null or not-null.

In his 2009 presentation "Null References: The Billion Dollar Mistake," Tony Hoare, the inventor of null, has this to say:

> I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

The problem with null values is that if you try to use a null value as a not-null value, you'll get an error of some kind. Because this null or not-null property is pervasive, it's extremely easy to make this kind of error.

However, the concept that null is trying to express is still a useful one: a null is a value that is currently invalid or absent for some reason.

The problem isn't really with the concept but with the particular implementation. As such, Rust does not have nulls, but it does have an enum that can encode the concept of a value being present or absent. This enum is `Option<T>`, and it is defined by the standard library as follows:

```
enum Option<T> {
    None,
    Some(T),
}
```

The `Option<T>` enum is so useful that it's even included in the prelude; you don't need to bring it into scope explicitly. Its variants are also included in the prelude: you can use `Some` and `None` directly without the `Option::` prefix. The `Option<T>` enum is still just a regular enum, and `Some(T)` and `None` are still variants of type `Option<T>`.

The `<T>` syntax is a feature of Rust we haven't talked about yet. It's a generic type parameter, and we'll cover generics in more detail in Chapter 10. For now, all you need to know is that `<T>` means

the `Some` variant of the `Option` enum can hold one piece of data of any type, and that each concrete type that gets used in place of `T` makes the overall `Option<T>` type a different type. Here are some examples of using `Option` values to hold number types and string types:

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

The type of `some_number` is `Option<i32>`. The type of `some_string` is `Option<&str>`, which is a different type. Rust can infer these types because we've specified a value inside the `Some` variant. For `absent_number`, Rust requires us to annotate the overall `Option` type: the compiler can't infer the type that the corresponding `Some` variant will hold by looking only at a `None` value. Here, we tell Rust that we mean for `absent_number` to be of type `Option<i32>`.

When we have a `Some` value, we know that a value is present and the value is held within the `Some`. When we have a `None` value, in some sense, it means the same thing as null: we don't have a valid value. So why is having `Option<T>` any better than having null?

In short, because `Option<T>` and `T` (where `T` can be any type) are different types, the compiler won't let us use an `Option<T>` value as if it were definitely a valid value. For example, this code won't compile because it's trying to add an `i8` to an `Option<i8>`:

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

If we run this code, we get an error message like this:

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

Intense! In effect, this error message means that Rust doesn't understand how to add an `i8` and an `Option<i8>`, because they're different types. When we have a value of a type like `i8` in Rust, the compiler will ensure that we always have a valid value. We can proceed confidently without having to check for null before using that value. Only when we have an `Option<i8>` (or whatever type of value we're working with) do we have to worry about possibly not having a value, and the compiler will make sure we handle that case before using the value.

In other words, you have to convert an `Option<T>` to a `T` before you can perform `T` operations with it. Generally, this helps catch one of the most common issues with null: assuming that something isn't null when it actually is.

Eliminating the risk of incorrectly assuming a not-null value helps you to be more confident in your code. In order to have a value that can possibly be null, you must explicitly opt in by making the type of that value `Option<T>`. Then, when you use that value, you are required to explicitly handle the case when the value is null. Everywhere that a value has a type that isn't an `Option<T>`, you *can* safely assume that the value isn't null. This was a deliberate design decision for Rust to limit null's pervasiveness and increase the safety of Rust code.

So, how do you get the `T` value out of a `Some` variant when you have a value of type `Option<T>` so you can use that value? The `Option<T>` enum has a large number of methods that are useful in a

variety of situations; you can check them out in its documentation. Becoming familiar with the methods on `Option<T>` will be extremely useful in your journey with Rust.

In general, in order to use an `Option<T>` value, you want to have code that will handle each variant. You want some code that will run only when you have a `Some(T)` value, and this code is allowed to use the inner `T`. You want some other code to run if you have a `None` value, and that code doesn't have a `T` value available. The `match` expression is a control flow construct that does just this when used with enums: it will run different code depending on which variant of the enum it has, and that code can use the data inside the matching value.

The `match` Control Flow Construct

Rust has an extremely powerful control flow construct called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards, and many other things; Chapter 18 covers all the different kinds of patterns and what they do. The power of `match` comes from the expressiveness of the patterns and the fact that the compiler confirms that all possible cases are handled.

Think of a `match` expression as being like a coin-sorting machine: coins slide down a track with variously sized holes along it, and each coin falls through the first hole it encounters that it fits into. In the same way, values go through each pattern in a `match`, and at the first pattern the value "fits," the value falls into the associated code block to be used during execution. Speaking of coins, let's use them as an example using `match`! We can write a function that takes an unknown United States coin and, in a similar way as the counting machine, determines which coin it is and return its value in cents, as shown here in Listing 6-3.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Listing 6-3: An enum and a `match` expression that has the variants of the enum as its patterns

Let's break down the `match` in the `value_in_cents` function. First, we list the `match` keyword followed by an expression, which in this case is the value `coin`. This seems very similar to an expression used with `if`, but there's a big difference: with `if`, the expression needs to return a Boolean value, but here, it can return any type. The type of `coin` in this example is the `Coin` enum that we defined on the first line.

Next are the `match` arms. An arm has two parts: a pattern and some code. The first arm here has a pattern that is the value `Coin::Penny` and then the `=>` operator that separates the pattern and the code to run. The code in this case is just the value `1`. Each arm is separated from the next with a comma.

When the `match` expression executes, it compares the resulting value against the pattern of each arm, in order. If a pattern matches the value, the code associated with that pattern is executed. If that pattern doesn't match the value, execution continues to the next arm, much as in a coin-sorting machine. We can have as many arms as we need: in Listing 6-3, our `match` has four arms.

The code associated with each arm is an expression, and the resulting value of the expression in the matching arm is the value that gets returned for the entire `match` expression.

We don't typically use curly brackets if the match arm code is short, as it is in Listing 6-3 where each arm just returns a value. If you want to run multiple lines of code in a match arm, you must use curly brackets. For example, the following code prints "Lucky penny!" every time the method is called with a `Coin::Penny`, but still returns the last value of the block, `1`:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

Another useful feature of match arms is that they can bind to the parts of the values that match the pattern. This is how we can extract values out of enum variants.

As an example, let's change one of our enum variants to hold data inside it. From 1999 through 2008, the United States minted quarters with different designs for each of the 50 states on one side. No other coins got state designs, so only quarters have this extra value. We can add this information to our `enum` by changing the `Quarter` variant to include a `UsState` value stored inside it, which we've done here in Listing 6-4.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Listing 6-4: A `Coin` enum in which the `Quarter` variant also holds a `UsState` value

Let's imagine that a friend is trying to collect all 50 state quarters. While we sort our loose change by coin type, we'll also call out the name of the state associated with each quarter so if it's one our friend doesn't have, they can add it to their collection.

In the match expression for this code, we add a variable called `state` to the pattern that matches values of the variant `Coin::Quarter`. When a `Coin::Quarter` matches, the `state` variable will bind to the value of that quarter's state. Then we can use `state` in the code for that arm, like so:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

If we were to call `value_in_cents(Coin::Quarter(UsState::Alaska))`, `coin` would be `Coin::Quarter(UsState::Alaska)`. When we compare that value with each of the match arms, none of them match until we reach `Coin::Quarter(state)`. At that point, the binding for `state` will be the value `UsState::Alaska`. We can then use that binding in the `println!` expression, thus getting the inner state value out of the `Coin` enum variant for `Quarter`.

Matching with `Option<T>`

In the previous section, we wanted to get the inner `T` value out of the `Some` case when using `Option<T>`; we can also handle `Option<T>` using `match` as we did with the `Coin` enum! Instead of comparing coins, we'll compare the variants of `Option<T>`, but the way that the `match` expression works remains the same.

Let's say we want to write a function that takes an `Option<i32>` and, if there's a value inside, adds 1 to that value. If there isn't a value inside, the function should return the `None` value and not attempt to perform any operations.

This function is very easy to write, thanks to `match`, and will look like Listing 6-5.

```
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
```

Listing 6-5: A function that uses a `match` expression on an `Option<i32>`

Let's examine the first execution of `plus_one` in more detail. When we call `plus_one(five)`, the variable `x` in the body of `plus_one` will have the value `Some(5)`. We then compare that against each match arm.

```
            None => None,
```

The `Some(5)` value doesn't match the pattern `None`, so we continue to the next arm.

```
            Some(i) => Some(i + 1),
```

Does `Some(5)` match `Some(i)`? Why yes it does! We have the same variant. The `i` binds to the value contained in `Some`, so `i` takes the value `5`. The code in the match arm is then executed, so we add 1 to the value of `i` and create a new `Some` value with our total `6` inside.

Now let's consider the second call of `plus_one` in Listing 6-5, where `x` is `None`. We enter the `match` and compare to the first arm.

```
            None => None,
```

It matches! There's no value to add to, so the program stops and returns the `None` value on the right side of `=>`. Because the first arm matched, no other arms are compared.

Combining `match` and enums is useful in many situations. You'll see this pattern a lot in Rust code: `match` against an enum, bind a variable to the data inside, and then execute code based on it. It's a bit tricky at first, but once you get used to it, you'll wish you had it in all languages. It's consistently a user favorite.

Matches Are Exhaustive

There's one other aspect of `match` we need to discuss. Consider this version of our `plus_one` function that has a bug and won't compile:

```
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
        }
    }
```

We didn't handle the `None` case, so this code will cause a bug. Luckily, it's a bug Rust knows how to catch. If we try to compile this code, we'll get this error:

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:3:15
    |
3   |         match x {
    |               ^ pattern `None` not covered
    |
    = help: ensure that all possible cases are being handled, possibly by adding
wildcards or more match arms
    = note: the matched value is of type `Option<i32>`

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error
```

Rust knows that we didn't cover every possible case and even knows which pattern we forgot! Matches in Rust are *exhaustive*: we must exhaust every last possibility in order for the code to be valid. Especially in the case of `Option<T>`, when Rust prevents us from forgetting to explicitly handle the `None` case, it protects us from assuming that we have a value when we might have null, thus making the billion-dollar mistake discussed earlier impossible.

Catch-all Patterns and the _ Placeholder

Using enums, we can also take special actions for a few particular values, but for all other values take one default action. Imagine we're implementing a game where, if you roll a 3 on a dice roll, your player doesn't move, but instead gets a new fancy hat. If you roll a 7, your player loses a fancy hat. For all other values, your player moves that number of spaces on the game board. Here's a `match` that implements that logic, with the result of the dice roll hardcoded rather than a random value, and all other logic represented by functions without bodies because actually implementing them is out of scope for this example:

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

For the first two arms, the patterns are the literal values 3 and 7. For the last arm that covers every other possible value, the pattern is the variable we've chosen to name `other`. The code that runs for the `other` arm uses the variable by passing it to the `move_player` function.

This code compiles, even though we haven't listed all the possible values a `u8` can have, because the last pattern will match all values not specifically listed. This catch-all pattern meets the requirement that `match` must be exhaustive. Note that we have to put the catch-all arm last because the patterns are evaluated in order. Rust will warn us if we add arms after a catch-all because those later arms would never match!

Rust also has a pattern we can use when we don't want to use the value in the catch-all pattern: `_`, which is a special pattern that matches any value and does not bind to that value. This tells Rust we

aren't going to use the value, so Rust won't warn us about an unused variable.

Let's change the rules of the game to be that if you roll anything other than a 3 or a 7, you must roll again. We don't need to use the value in that case, so we can change our code to use `_` instead of the variable named `other`:

```
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => reroll(),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn reroll() {}
```

This example also meets the exhaustiveness requirement because we're explicitly ignoring all other values in the last arm; we haven't forgotten anything.

If we change the rules of the game one more time, so that nothing else happens on your turn if you roll anything other than a 3 or a 7, we can express that by using the unit value (the empty tuple type we mentioned in "The Tuple Type" section) as the code that goes with the `_` arm:

```
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
```

Here, we're telling Rust explicitly that we aren't going to use any other value that doesn't match a pattern in an earlier arm, and we don't want to run any code in this case.

There's more about patterns and matching that we'll cover in Chapter 18. For now, we're going to move on to the `if let` syntax, which can be useful in situations where the `match` expression is a bit wordy.

Concise Control Flow with `if let`

The `if let` syntax lets you combine `if` and `let` into a less verbose way to handle values that match one pattern while ignoring the rest. Consider the program in Listing 6-6 that matches on an `Option<u8>` value in the `config_max` variable but only wants to execute code if the value is the `Some` variant.

```
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {}", max),
        _ => (),
    }
```

Listing 6-6: A `match` that only cares about executing code when the value is `Some`

If the value is `Some`, we print out the value in the `Some` variant by binding the value to the variable `max` in the pattern. We don't want to do anything with the `None` value. To satisfy the `match` expression, we have to add `_ => ()` after processing just one variant, which is annoying boilerplate code to add.

Instead, we could write this in a shorter way using `if let`. The following code behaves the same as the `match` in Listing 6-6:

```
    let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {}", max);
    }
```

The syntax `if let` takes a pattern and an expression separated by an equal sign. It works the same way as a `match`, where the expression is given to the `match` and the pattern is its first arm. In this case, the pattern is `Some(max)`, and the `max` binds to the value inside the `Some`. We can then use

`max` in the body of the `if let` block in the same way as we used `max` in the corresponding `match` arm. The code in the `if let` block isn't run if the value doesn't match the pattern.

Using `if let` means less typing, less indentation, and less boilerplate code. However, you lose the exhaustive checking that `match` enforces. Choosing between `match` and `if let` depends on what you're doing in your particular situation and whether gaining conciseness is an appropriate trade-off for losing exhaustive checking.

In other words, you can think of `if let` as syntax sugar for a `match` that runs code when the value matches one pattern and then ignores all other values.

We can include an `else` with an `if let`. The block of code that goes with the `else` is the same as the block of code that would go with the `_` case in the `match` expression that is equivalent to the `if let` and `else`. Recall the `Coin` enum definition in Listing 6-4, where the `Quarter` variant also held a `UsState` value. If we wanted to count all non-quarter coins we see while also announcing the state of the quarters, we could do that with a `match` expression like this:

```
    let mut count = 0;
    match coin {
        Coin::Quarter(state) => println!("State quarter from {:?}!", state),
        _ => count += 1,
    }
```

Or we could use an `if let` and `else` expression like this:

```
    let mut count = 0;
    if let Coin::Quarter(state) = coin {
        println!("State quarter from {:?}!", state);
    } else {
        count += 1;
    }
```

If you have a situation in which your program has logic that is too verbose to express using a `match`, remember that `if let` is in your Rust toolbox as well.

Summary

We've now covered how to use enums to create custom types that can be one of a set of enumerated values. We've shown how the standard library's `Option<T>` type helps you use the type system to prevent errors. When enum values have data inside them, you can use `match` or `if let` to extract and use those values, depending on how many cases you need to handle.

Your Rust programs can now express concepts in your domain using structs and enums. Creating custom types to use in your API ensures type safety: the compiler will make certain your functions get only values of the type each function expects.

In order to provide a well-organized API to your users that is straightforward to use and only exposes exactly what your users will need, let's now turn to Rust's modules.

Managing Growing Projects with Packages, Crates, and Modules

As you write large programs, organizing your code will be important because keeping track of your entire program in your head will become impossible. By grouping related functionality and separating code with distinct features, you'll clarify where to find code that implements a particular feature and where to go to change how a feature works.

The programs we've written so far have been in one module in one file. As a project grows, you can organize code by splitting it into multiple modules and then multiple files. A package can contain multiple binary crates and optionally one library crate. As a package grows, you can extract parts into separate crates that become external dependencies. This chapter covers all these techniques. For very large projects of a set of interrelated packages that evolve together, Cargo provides workspaces, which we'll cover in the "Cargo Workspaces" section in Chapter 14.

In addition to grouping functionality, encapsulating implementation details lets you reuse code at a higher level: once you've implemented an operation, other code can call that code via the code's public interface without knowing how the implementation works. The way you write code defines which parts are public for other code to use and which parts are private implementation details that you reserve the right to change. This is another way to limit the amount of detail you have to keep in your head.

A related concept is scope: the nested context in which code is written has a set of names that are defined as "in scope." When reading, writing, and compiling code, programmers and compilers need to know whether a particular name at a particular spot refers to a variable, function, struct, enum, module, constant, or other item and what that item means. You can create scopes and change which names are in or out of scope. You can't have two items with the same name in the same scope; tools are available to resolve name conflicts.

Rust has a number of features that allow you to manage your code's organization, including which details are exposed, which details are private, and what names are in each scope in your programs. These features, sometimes collectively referred to as the *module system*, include:

- **Packages:** A Cargo feature that lets you build, test, and share crates
- **Crates:** A tree of modules that produces a library or executable
- **Modules** and **use:** Let you control the organization, scope, and privacy of paths
- **Paths:** A way of naming an item, such as a struct, function, or module

In this chapter, we'll cover all these features, discuss how they interact, and explain how to use them to manage scope. By the end, you should have a solid understanding of the module system and be able to work with scopes like a pro!

Packages and Crates

The first parts of the module system we'll cover are packages and crates.

A *package* is one or more crates that provide a set of functionality. A package contains a *Cargo.toml* file that describes how to build those crates.

A *crate* can be a binary crate or a library crate. *Binary crates* are programs you can compile to an executable that you can run, such as a command-line program or a server. They must have a function called `main` that defines what happens when the executable runs. All the crates we've created so far have been binary crates.

Library crates don't have a `main` function, and they don't compile to an executable. They define functionality intended to be shared with multiple projects. For example, the `rand` crate we used in Chapter 2 provides functionality that generates random numbers.

The *crate root* is a source file that the Rust compiler starts from and makes up the root module of your crate (we'll explain modules in depth in the "Defining Modules to Control Scope and Privacy" section).

Several rules determine what a package can contain. A package can contain at most one library crate. It can contain as many binary crates as you'd like, but it must contain at least one crate (either library or binary).

Let's walk through what happens when we create a package. First, we enter the command `cargo new`:

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

When we entered the command, Cargo created a *Cargo.toml* file, giving us a package. Looking at the contents of *Cargo.toml*, there's no mention of *src/main.rs* because Cargo follows a convention that *src/main.rs* is the crate root of a binary crate with the same name as the package. Likewise, Cargo knows that if the package directory contains *src/lib.rs*, the package contains a library crate with the same name as the package, and *src/lib.rs* is its crate root. Cargo passes the crate root files to `rustc` to build the library or binary.

Here, we have a package that only contains *src/main.rs*, meaning it only contains a binary crate named `my-project`. If a package contains *src/main.rs* and *src/lib.rs*, it has two crates: a binary and a library, both with the same name as the package. A package can have multiple binary crates by placing files in the *src/bin* directory: each file will be a separate binary crate.

Defining Modules to Control Scope and Privacy

In this section, we'll talk about modules and other parts of the module system, namely *paths* that allow you to name items; the `use` keyword that brings a path into scope; and the `pub` keyword to make items public. We'll also discuss the `as` keyword, external packages, and the glob operator.

First, we're going to start with a list of rules for easy reference when you're organizing your code in the future. Then we'll explain each of the rules in detail.

Modules Quick Reference

Here's how modules, paths, the `use` keyword, and the `pub` keyword work in the compiler, and how most developers organize their code. We'll be going through examples of each of these rules, but this is a great place to look in the future as a reminder of how modules work.

- **Start from the crate root**: When compiling a crate, the compiler first looks in the crate root file (usually *src/lib.rs* for a library crate or *src/main.rs* for a binary crate).
- **Declaring modules**: In the crate root file, you can declare a new module named, say, "garden", with `mod garden;`. The compiler will look for the code inside the module in these places:
  - Inline, directly following `mod garden`, within curly brackets instead of the semicolon
  - In the file *src/garden.rs*
  - In the file *src/garden/mod.rs*
- **Declaring submodules**: In any file other than the crate root that's being compiled as part of the crate (for example, *src/garden.rs*), you can declare submodules (for example, `mod vegetables;`). The compiler will look for the code inside submodules in these places within a directory named for the parent module:
  - Inline, directly following `mod vegetables`, within curly brackets instead of the semicolon
  - In the file *src/garden/vegetables.rs*
  - In the file *src/garden/vegetables/mod.rs*

- **Paths to code in modules**: Once a module is being compiled as part of your crate, you can refer to code in that module (for example, an `Asparagus` type in the garden vegetables module) from anywhere else in this crate by using the path `crate::garden::vegetables::Asparagus` as long as the privacy rules allow.
- **Private vs public**: Code within a module is private from its parent modules by default. To make a module public, declare it with `pub mod` instead of `mod`. To make items within a public module public as well, use `pub` before their declarations.
- **The `use` keyword**: Within a scope, the `use` keyword creates shortcuts to items to reduce repetition of long paths. In any scope that can refer to `crate::garden::vegetables::Asparagus`, you can create a shortcut with `use crate::garden::vegetables::Asparagus;` and then only need to write `Asparagus` to make use of that type in the scope.

Here's a binary crate named `backyard` that illustrates these rules. The crate's directory, also named `backyard`, contains these files and directories:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

The crate root file, in this case *src/main.rs*, contains:

Filename: src/main.rs

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

The `pub mod garden;` means the compiler includes the code it finds in *src/garden.rs*, which is:

Filename: src/garden.rs

```
pub mod vegetables;
```

And `pub mod vegetables;` means the code in *src/garden/vegetables.rs* is included too:

```
#[derive(Debug)]
pub struct Asparagus {}
```

Now let's get into the details of these rules and demonstrate them in action!

Grouping Related Code in Modules

Modules let us organize code within a crate into groups for readability and easy reuse. Modules also control the *privacy* of items, which is whether an item can be used by outside code (*public*) or is an internal implementation detail and not available for outside use (*private*).

As an example, let's write a library crate that provides the functionality of a restaurant. We'll define the signatures of functions but leave their bodies empty to concentrate on the organization of the code, rather than actually implement a restaurant in code.

In the restaurant industry, some parts of a restaurant are referred to as *front of house* and others as *back of house*. Front of house is where customers are; this is where hosts seat customers, servers take orders and payment, and bartenders make drinks. Back of house is where the chefs and cooks work in the kitchen, dishwashers clean up, and managers do administrative work.

To structure our crate in the same way that a real restaurant works, we can organize the functions into nested modules. Create a new library named `restaurant` by running `cargo new --lib restaurant`; then put the code in Listing 7-1 into *src/lib.rs* to define some modules and function signatures.

Filename: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Listing 7-1: A `front_of_house` module containing other modules that then contain functions

We define a module by starting with the `mod` keyword and then specify the name of the module (in this case, `front_of_house`) and place curly brackets around the body of the module. Inside modules, we can have other modules, as in this case with the modules `hosting` and `serving`.

Modules can also hold definitions for other items, such as structs, enums, constants, traits, or—as in Listing 7-1—functions.

By using modules, we can group related definitions together and name why they're related. Programmers using this code would have an easier time finding the definitions they wanted to use because they could navigate the code based on the groups rather than having to read through all the definitions. Programmers adding new functionality to this code would know where to place the code to keep the program organized.

Earlier, we mentioned that *src/main.rs* and *src/lib.rs* are called crate roots. The reason for their name is that the contents of either of these two files form a module named `crate` at the root of the crate's module structure, known as the *module tree*.

Listing 7-2 shows the module tree for the structure in Listing 7-1.

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Listing 7-2: The module tree for the code in Listing 7-1

This tree shows how some of the modules nest inside one another (for example, `hosting` nests inside `front_of_house`). The tree also shows that some modules are *siblings* to each other, meaning they're defined in the same module (`hosting` and `serving` are defined within `front_of_house`). To continue the family metaphor, if module A is contained inside module B, we say that module A is the *child* of module B and that module B is the *parent* of module A. Notice that the entire module tree is rooted under the implicit module named `crate`.

The module tree might remind you of the filesystem's directory tree on your computer; this is a very apt comparison! Just like directories in a filesystem, you use modules to organize your code. And just like files in a directory, we need a way to find our modules.

Paths for Referring to an Item in the Module Tree

To show Rust where to find an item in a module tree, we use a path in the same way we use a path when navigating a filesystem. If we want to call a function, we need to know its path.

A path can take two forms:

- An *absolute path* starts from a crate root by using a crate name (for code from an external crate) or a literal `crate` (for code from the current crate).
- A *relative path* starts from the current module and uses `self`, `super`, or an identifier in the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (`::`).

Let's return to the example in Listing 7-1. How do we call the `add_to_waitlist` function? This is the same as asking, what's the path of the `add_to_waitlist` function? Listing 7-3 contains Listing 7-1 with some of the modules and functions removed. We'll show two ways to call the `add_to_waitlist` function from a new function `eat_at_restaurant` defined in the crate root. The `eat_at_restaurant` function is part of our library crate's public API, so we mark it with the `pub` keyword. In the "Exposing Paths with the `pub` Keyword" section, we'll go into more detail about `pub`. Note that this example won't compile just yet; we'll explain why in a bit.

Filename: src/lib.rs

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Listing 7-3: Calling the `add_to_waitlist` function using absolute and relative paths

The first time we call the `add_to_waitlist` function in `eat_at_restaurant`, we use an absolute path. The `add_to_waitlist` function is defined in the same crate as `eat_at_restaurant`, which means we can use the `crate` keyword to start an absolute path.

After `crate`, we include each of the successive modules until we make our way to `add_to_waitlist`. You can imagine a filesystem with the same structure, and we'd specify the path `/front_of_house/hosting/add_to_waitlist` to run the `add_to_waitlist` program; using the `crate` name to start from the crate root is like using `/` to start from the filesystem root in your shell.

The second time we call `add_to_waitlist` in `eat_at_restaurant`, we use a relative path. The path starts with `front_of_house`, the name of the module defined at the same level of the module tree as `eat_at_restaurant`. Here the filesystem equivalent would be using the path `front_of_house/hosting/add_to_waitlist`. Starting with a name means that the path is relative.

Choosing whether to use a relative or absolute path is a decision you'll make based on your project. The decision should depend on whether you're more likely to move item definition code separately from or together with the code that uses the item. For example, if we move the `front_of_house`

module and the `eat_at_restaurant` function into a module named `customer_experience`, we'd need to update the absolute path to `add_to_waitlist`, but the relative path would still be valid. However, if we moved the `eat_at_restaurant` function separately into a module named `dining`, the absolute path to the `add_to_waitlist` call would stay the same, but the relative path would need to be updated. Our preference is to specify absolute paths because it's more likely we'll want to move code definitions and item calls independently of each other.

Let's try to compile Listing 7-3 and find out why it won't compile yet! The error we get is shown in Listing 7-4.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: module `hosting` is private
 --> src/lib.rs:9:28
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                            ^^^^^^^ private module
  |
note: the module `hosting` is defined here
 --> src/lib.rs:2:5
  |
2 |     mod hosting {
  |     ^^^^^^^^^^^

error[E0603]: module `hosting` is private
  --> src/lib.rs:12:21
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                     ^^^^^^^ private module
   |
note: the module `hosting` is defined here
  --> src/lib.rs:2:5
   |
2  |     mod hosting {
   |     ^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Listing 7-4: Compiler errors from building the code in Listing 7-3

The error messages say that module `hosting` is private. In other words, we have the correct paths for the `hosting` module and the `add_to_waitlist` function, but Rust won't let us use them because it doesn't have access to the private sections.

Modules aren't useful only for organizing your code. They also define Rust's *privacy boundary*: the line that encapsulates the implementation details external code isn't allowed to know about, call, or

rely on. So, if you want to make an item like a function or struct private, you put it in a module.

The way privacy works in Rust is that all items (functions, methods, structs, enums, modules, and constants) are private by default. Items in a parent module can't use the private items inside child modules, but items in child modules can use the items in their ancestor modules. The reason is that child modules wrap and hide their implementation details, but the child modules can see the context in which they're defined. To continue with the restaurant metaphor, think of the privacy rules as being like the back office of a restaurant: what goes on in there is private to restaurant customers, but office managers can see and do everything in the restaurant in which they operate.

Rust chose to have the module system function this way so that hiding inner implementation details is the default. That way, you know which parts of the inner code you can change without breaking outer code. But you can expose inner parts of child modules' code to outer ancestor modules by using the `pub` keyword to make an item public.

Exposing Paths with the `pub` Keyword

Let's return to the error in Listing 7-4 that told us the `hosting` module is private. We want the `eat_at_restaurant` function in the parent module to have access to the `add_to_waitlist` function in the child module, so we mark the `hosting` module with the `pub` keyword, as shown in Listing 7-5.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Listing 7-5: Declaring the `hosting` module as `pub` to use it from `eat_at_restaurant`

Unfortunately, the code in Listing 7-5 still results in an error, as shown in Listing 7-6.

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0603]: function `add_to_waitlist` is private
 --> src/lib.rs:9:37
  |
9 |     crate::front_of_house::hosting::add_to_waitlist();
  |                                     ^^^^^^^^^^^^^^^ private function
  |
note: the function `add_to_waitlist` is defined here
 --> src/lib.rs:3:9
  |
3 |         fn add_to_waitlist() {}
  |         ^^^^^^^^^^^^^^^^^^^^

error[E0603]: function `add_to_waitlist` is private
  --> src/lib.rs:12:30
   |
12 |     front_of_house::hosting::add_to_waitlist();
   |                              ^^^^^^^^^^^^^^^ private function
   |
note: the function `add_to_waitlist` is defined here
  --> src/lib.rs:3:9
   |
3  |         fn add_to_waitlist() {}
   |         ^^^^^^^^^^^^^^^^^^^^

For more information about this error, try `rustc --explain E0603`.
error: could not compile `restaurant` due to 2 previous errors
```

Listing 7-6: Compiler errors from building the code in Listing 7-5

What happened? Adding the `pub` keyword in front of `mod hosting` makes the module public. With this change, if we can access `front_of_house`, we can access `hosting`. But the *contents* of `hosting` are still private; making the module public doesn't make its contents public. The `pub` keyword on a module only lets code in its ancestor modules refer to it.

The errors in Listing 7-6 say that the `add_to_waitlist` function is private. The privacy rules apply to structs, enums, functions, and methods as well as modules.

Let's also make the `add_to_waitlist` function public by adding the `pub` keyword before its definition, as in Listing 7-7.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

Listing 7-7: Adding the `pub` keyword to `mod hosting` and `fn add_to_waitlist` lets us call the function from `eat_at_restaurant`

Now the code will compile! Let's look at the absolute and the relative path and double-check why adding the `pub` keyword lets us use these paths in `add_to_waitlist` with respect to the privacy rules.

In the absolute path, we start with `crate`, the root of our crate's module tree. Then the `front_of_house` module is defined in the crate root. The `front_of_house` module isn't public, but because the `eat_at_restaurant` function is defined in the same module as `front_of_house` (that is, `eat_at_restaurant` and `front_of_house` are siblings), we can refer to `front_of_house` from `eat_at_restaurant`. Next is the `hosting` module marked with `pub`. We can access the parent

module of `hosting`, so we can access `hosting`. Finally, the `add_to_waitlist` function is marked with `pub` and we can access its parent module, so this function call works!

In the relative path, the logic is the same as the absolute path except for the first step: rather than starting from the crate root, the path starts from `front_of_house`. The `front_of_house` module is defined within the same module as `eat_at_restaurant`, so the relative path starting from the module in which `eat_at_restaurant` is defined works. Then, because `hosting` and `add_to_waitlist` are marked with `pub`, the rest of the path works, and this function call is valid!

If you plan on sharing your library crate so other projects can use your code, your public API is your contract with users of your crate about how they interact with your code. There are many considerations around managing changes to your public API to make it easier for people to depend on your crate. These considerations are out of the scope of this book; if you're interested in this topic, see The Rust API Guidelines.

Best Practices for Packages with a Binary and a Library

We mentioned a package can contain both a *src/main.rs* binary crate root as well as a *src/lib.rs* library crate root, and both crates will have the package name by default. Typically, packages with this pattern will have just enough code in the binary crate to start an executable that calls code with the library crate. This lets other projects benefit from the most functionality that the package provides, because the library crate's code can be shared.

The module tree should be defined in *src/lib.rs*. Then, any public items can be used in the binary crate by starting paths with the name of the package. The binary crate becomes a user of the library crate just like a completely external crate would use the library crate: it can only use the public API. This helps you design a good API; not only are you the author, you're also a client!

In Chapter 12, we'll demonstrate this organizational practice with a command-line program that will contain both a binary crate and a library crate.

Starting Relative Paths with `super`

We can also construct relative paths that begin in the parent module by using `super` at the start of the path. This is like starting a filesystem path with the `..` syntax. Why would we want to do this?

Consider the code in Listing 7-8 that models the situation in which a chef fixes an incorrect order and personally brings it out to the customer. The function `fix_incorrect_order` defined in the `back_of_house` module calls the function `deliver_order` defined in the parent module by specifying the path to `deliver_order` starting with `super`:

Filename: src/lib.rs

```
fn deliver_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::deliver_order();
    }

    fn cook_order() {}
}
```

Listing 7-8: Calling a function using a relative path starting with `super`

The `fix_incorrect_order` function is in the `back_of_house` module, so we can use `super` to go to the parent module of `back_of_house`, which in this case is `crate`, the root. From there, we look for `deliver_order` and find it. Success! We think the `back_of_house` module and the `deliver_order`

function are likely to stay in the same relationship to each other and get moved together should we decide to reorganize the crate's module tree. Therefore, we used `super` so we'll have fewer places to update code in the future if this code gets moved to a different module.

Making Structs and Enums Public

We can also use `pub` to designate structs and enums as public, but there are a few extra details. If we use `pub` before a struct definition, we make the struct public, but the struct's fields will still be private. We can make each field public or not on a case-by-case basis. In Listing 7-9, we've defined a public `back_of_house::Breakfast` struct with a public `toast` field but a private `seasonal_fruit` field. This models the case in a restaurant where the customer can pick the type of bread that comes with a meal, but the chef decides which fruit accompanies the meal based on what's in season and in stock. The available fruit changes quickly, so customers can't choose the fruit or even see which fruit they'll get.

Filename: src/lib.rs

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    // Order a breakfast in the summer with Rye toast
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);

    // The next line won't compile if we uncomment it; we're not allowed
    // to see or modify the seasonal fruit that comes with the meal
    // meal.seasonal_fruit = String::from("blueberries");
}
```

Listing 7-9: A struct with some public fields and some private fields

Because the `toast` field in the `back_of_house::Breakfast` struct is public, in `eat_at_restaurant` we can write and read to the `toast` field using dot notation. Notice that we can't use the `seasonal_fruit` field in `eat_at_restaurant` because `seasonal_fruit` is private. Try uncommenting the line modifying the `seasonal_fruit` field value to see what error you get!

Also, note that because `back_of_house::Breakfast` has a private field, the struct needs to provide a public associated function that constructs an instance of `Breakfast` (we've named it `summer` here). If `Breakfast` didn't have such a function, we couldn't create an instance of `Breakfast` in `eat_at_restaurant` because we couldn't set the value of the private `seasonal_fruit` field in `eat_at_restaurant`.

In contrast, if we make an enum public, all of its variants are then public. We only need the `pub` before the `enum` keyword, as shown in Listing 7-10.

Filename: src/lib.rs

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house::Appetizer::Soup;
    let order2 = back_of_house::Appetizer::Salad;
}
```

Listing 7-10: Designating an enum as public makes all its variants public

Because we made the `Appetizer` enum public, we can use the `Soup` and `Salad` variants in `eat_at_restaurant`. Enums aren't very useful unless their variants are public; it would be annoying to have to annotate all enum variants with `pub` in every case, so the default for enum variants is to be public. Structs are often useful without their fields being public, so struct fields follow the general rule of everything being private by default unless annotated with `pub`.

There's one more situation involving `pub` that we haven't covered, and that is our last module system feature: the `use` keyword. We'll cover `use` by itself first, and then we'll show how to

combine `pub` and `use`.

Bringing Paths into Scope with the `use` Keyword

It might seem like the paths we've written to call functions so far are inconveniently long and repetitive. For example, in Listing 7-7, whether we chose the absolute or relative path to the `add_to_waitlist` function, every time we wanted to call `add_to_waitlist` we had to specify `front_of_house` and `hosting` too. Fortunately, there's a way to simplify this process. We can create a shortcut to a path with the `use` keyword once, and then use the shorter name everywhere else in the scope.

In Listing 7-11, we bring the `crate::front_of_house::hosting` module into the scope of the `eat_at_restaurant` function so we only have to specify `hosting::add_to_waitlist` to call the `add_to_waitlist` function in `eat_at_restaurant`.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Listing 7-11: Bringing a module into scope with `use`

Adding `use` and a path in a scope is similar to creating a symbolic link in the filesystem. By adding `use crate::front_of_house::hosting` in the crate root, `hosting` is now a valid name in that scope, just as though the `hosting` module had been defined in the crate root. Paths brought into scope with `use` also check privacy, like any other paths.

Note that `use` only creates the shortcut for the particular scope in which the `use` occurs. Listing 7-12 moves the `eat_at_restaurant` function into a new child module named `customer`, which is then a different scope than the `use` statement and the function body won't compile:

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting;

mod customer {
    pub fn eat_at_restaurant() {
        hosting::add_to_waitlist();
    }
}
```

Listing 7-12: A `use` statement only applies in the scope it's in

The compiler error shows that the shortcut no longer applies within the `customer` module:

```
$ cargo build
   Compiling restaurant v0.1.0 (file:///projects/restaurant)
error[E0433]: failed to resolve: use of undeclared crate or module `hosting`
  --> src/lib.rs:11:9
   |
11 |         hosting::add_to_waitlist();
   |         ^^^^^^^ use of undeclared crate or module `hosting`

warning: unused import: `crate::front_of_house::hosting`
 --> src/lib.rs:7:5
  |
7 | use crate::front_of_house::hosting;
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

For more information about this error, try `rustc --explain E0433`.
warning: `restaurant` (lib) generated 1 warning
error: could not compile `restaurant` due to previous error; 1 warning emitted
```

Notice there's also a warning that the `use` is no longer used in its scope! To fix this problem, move the `use` within the `customer` module too, or reference the shortcut in the parent module with `super::hosting` within the child `customer` module.

Creating Idiomatic `use` Paths

In Listing 7-11, you might have wondered why we specified `use crate::front_of_house::hosting` and then called `hosting::add_to_waitlist` in `eat_at_restaurant` rather than specifying the `use` path all the way out to the `add_to_waitlist` function to achieve the same result, as in Listing 7-13.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

use crate::front_of_house::hosting::add_to_waitlist;

pub fn eat_at_restaurant() {
    add_to_waitlist();
}
```

Listing 7-13: Bringing the `add_to_waitlist` function into scope with `use`, which is unidiomatic

Although both Listing 7-11 and 7-13 accomplish the same task, Listing 7-11 is the idiomatic way to bring a function into scope with `use`. Bringing the function's parent module into scope with `use` means we have to specify the parent module when calling the function. Specifying the parent module when calling the function makes it clear that the function isn't locally defined while still minimizing repetition of the full path. The code in Listing 7-13 is unclear as to where `add_to_waitlist` is defined.

On the other hand, when bringing in structs, enums, and other items with `use`, it's idiomatic to specify the full path. Listing 7-14 shows the idiomatic way to bring the standard library's `HashMap` struct into the scope of a binary crate.

Filename: src/main.rs

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Listing 7-14: Bringing `HashMap` into scope in an idiomatic way

There's no strong reason behind this idiom: it's just the convention that has emerged, and folks have gotten used to reading and writing Rust code this way.

The exception to this idiom is if we're bringing two items with the same name into scope with `use` statements, because Rust doesn't allow that. Listing 7-15 shows how to bring two `Result` types into scope that have the same name but different parent modules and how to refer to them.

Filename: src/lib.rs

```
use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
}

fn function2() -> io::Result<()> {
    // --snip--
}
```

Listing 7-15: Bringing two types with the same name into the same scope requires using their parent modules.

As you can see, using the parent modules distinguishes the two `Result` types. If instead we specified `use std::fmt::Result` and `use std::io::Result`, we'd have two `Result` types in the same scope and Rust wouldn't know which one we meant when we used `Result`.

Providing New Names with the `as` Keyword

There's another solution to the problem of bringing two types of the same name into the same scope with `use`: after the path, we can specify `as` and a new local name, or alias, for the type.

Listing 7-16 shows another way to write the code in Listing 7-15 by renaming one of the two `Result` types using `as`.

Filename: src/lib.rs

```
use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

Listing 7-16: Renaming a type when it's brought into scope with the `as` keyword

In the second `use` statement, we chose the new name `IoResult` for the `std::io::Result` type, which won't conflict with the `Result` from `std::fmt` that we've also brought into scope. Listing 7-15 and Listing 7-16 are considered idiomatic, so the choice is up to you!

Re-exporting Names with `pub use`

When we bring a name into scope with the `use` keyword, the name available in the new scope is private. To enable the code that calls our code to refer to that name as if it had been defined in that code's scope, we can combine `pub` and `use`. This technique is called *re-exporting* because we're bringing an item into scope but also making that item available for others to bring into their scope.

Listing 7-17 shows the code in Listing 7-11 with `use` in the root module changed to `pub use`.

Filename: src/lib.rs

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Listing 7-17: Making a name available for any code to use from a new scope with `pub use`

Before this change, external code would have to call the `add_to_waitlist` function by using the path `restaurant::front_of_house::hosting::add_to_waitlist()`. Now that this `pub use` has re-exported the `hosting` module from the root module, external code can now use the path `restaurant::hosting::add_to_waitlist()` instead.

Re-exporting is useful when the internal structure of your code is different from how programmers calling your code would think about the domain. For example, in this restaurant metaphor, the people running the restaurant think about "front of house" and "back of house." But customers visiting a restaurant probably won't think about the parts of the restaurant in those terms. With `pub use`, we can write our code with one structure but expose a different structure. Doing so makes our library well organized for programmers working on the library and programmers calling the library. We'll look at another example of `pub use` and how it affects your crate's documentation in the "Exporting a Convenient Public API with `pub use`" section of Chapter 14.

Using External Packages

In Chapter 2, we programmed a guessing game project that used an external package called `rand` to get random numbers. To use `rand` in our project, we added this line to *Cargo.toml*:

Filename: Cargo.toml

```
rand = "0.8.3"
```

Adding `rand` as a dependency in *Cargo.toml* tells Cargo to download the `rand` package and any dependencies from crates.io and make `rand` available to our project.

Then, to bring `rand` definitions into the scope of our package, we added a `use` line starting with the name of the crate, `rand`, and listed the items we wanted to bring into scope. Recall that in the "Generating a Random Number" section in Chapter 2, we brought the `Rng` trait into scope and called the `rand::thread_rng` function:

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1..=100);
}
```

Members of the Rust community have made many packages available at crates.io, and pulling any of them into your package involves these same steps: listing them in your package's *Cargo.toml* file and using `use` to bring items from their crates into scope.

Note that the standard library (`std`) is also a crate that's external to our package. Because the standard library is shipped with the Rust language, we don't need to change *Cargo.toml* to include `std`. But we do need to refer to it with `use` to bring items from there into our package's scope. For example, with `HashMap` we would use this line:

```

use std::collections::HashMap;
```

This is an absolute path starting with `std`, the name of the standard library crate.

Using Nested Paths to Clean Up Large `use` Lists

If we're using multiple items defined in the same crate or same module, listing each item on its own line can take up a lot of vertical space in our files. For example, these two `use` statements we had in the Guessing Game in Listing 2-4 bring items from `std` into scope:

Filename: src/main.rs

```
// --snip--
use std::cmp::Ordering;
use std::io;
// --snip--
```

Instead, we can use nested paths to bring the same items into scope in one line. We do this by specifying the common part of the path, followed by two colons, and then curly brackets around a list of the parts of the paths that differ, as shown in Listing 7-18.

Filename: src/main.rs

```
// --snip--
use std::{cmp::Ordering, io};
// --snip--
```

Listing 7-18: Specifying a nested path to bring multiple items with the same prefix into scope

In bigger programs, bringing many items into scope from the same crate or module using nested paths can reduce the number of separate `use` statements needed by a lot!

We can use a nested path at any level in a path, which is useful when combining two `use` statements that share a subpath. For example, Listing 7-19 shows two `use` statements: one that brings `std::io` into scope and one that brings `std::io::Write` into scope.

Filename: src/lib.rs

```
use std::io;
use std::io::Write;
```

Listing 7-19: Two `use` statements where one is a subpath of the other

The common part of these two paths is `std::io`, and that's the complete first path. To merge these two paths into one `use` statement, we can use `self` in the nested path, as shown in Listing 7-20.

Filename: src/lib.rs

```
use std::io::{self, Write};
```

Listing 7-20: Combining the paths in Listing 7-19 into one `use` statement

This line brings `std::io` and `std::io::Write` into scope.

The Glob Operator

If we want to bring *all* public items defined in a path into scope, we can specify that path followed by `*`, the glob operator:

```
use std::collections::*;
```

This `use` statement brings all public items defined in `std::collections` into the current scope. Be careful when using the glob operator! Glob can make it harder to tell what names are in scope and where a name used in your program was defined.

The glob operator is often used when testing to bring everything under test into the `tests` module; we'll talk about that in the "How to Write Tests" section in Chapter 11. The glob operator is also

sometimes used as part of the prelude pattern: see the standard library documentation for more information on that pattern.

Separating Modules into Different Files

So far, all the examples in this chapter defined multiple modules in one file. When modules get large, you might want to move their definitions to a separate file to make the code easier to navigate.

For example, let's start from the code in Listing 7-17 and extract modules into files instead of having all the modules defined in the crate root file. In this case, the crate root file is *src/lib.rs*, but this procedure also works with binary crates whose crate root file is *src/main.rs*.

First, we'll extract the `front_of_house` module to its own file. Remove the code inside the curly brackets for the `front_of_house` module, leaving only the `mod front_of_house;` declaration, so that *src/lib.rs* contains the code shown in Listing 7-21. Note that this won't compile until we create the *src/front_of_house.rs* file in Listing 7-22.

Filename: src/lib.rs

```
mod front_of_house;

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
}
```

Listing 7-21: Declaring the `front_of_house` module whose body will be in *src/front_of_house.rs*

Next, place the code that was in the curly brackets into a new file named *src/front_of_house.rs*, as shown in Listing 7-22. The compiler knows to look in this file because of the module declaration it found in the crate root with the name `front_of_house`.

Filename: src/front_of_house.rs

```
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

Listing 7-22: Definitions inside the `front_of_house` module in *src/front_of_house.rs*

Note that you only need to load the contents of a file using a `mod` declaration once somewhere in your module tree. Once the compiler knows the file is part of the project (and knows where in the module tree the code resides because of where you've put the `mod` statement), other files in your project should refer to the code in that file using a path to where it was declared as covered in the "Paths for Referring to an Item in the Module Tree" section. In other words, `mod` is *not* an "include" operation that other programming languages have.

Next, we'll extract the `hosting` module to its own file as well. The process is a bit different because `hosting` is a child module of `front_of_house`, not of the root module. The file for `hosting` will be in a directory named for its place in the module tree.

To start moving `hosting`, we change *src/front_of_house.rs* to contain only the declaration of the `hosting` module:

Filename: src/front_of_house.rs

```
pub mod hosting;
```

Then we create a *src/front_of_house* directory and a file *src/front_of_house/hosting.rs* to contain the definitions made in the `hosting` module:

Filename: src/front_of_house/hosting.rs

```
pub fn add_to_waitlist() {}
```

If we instead put *hosting.rs* in the *src* directory, the compiler would expect that code to be in a `hosting` module declared in the crate root, not as a child of the `front_of_house` module. The rules the compiler follows to know what files to look in for modules' code means the directories and files more closely match the module tree.

Alternate File Paths

This section covered the most idiomatic file paths the Rust compiler uses; but an older file path is also still supported.

For a module named `front_of_house` declared in the crate root, the compiler will look for the module's code in:

- *src/front_of_house.rs* (what we covered)
- *src/front_of_house/mod.rs* (older, still supported path)

For a module named `hosting` that is a submodule of `front_of_house`, the compiler will look for the module's code in:

- *src/front_of_house/hosting.rs* (what we covered)
- *src/front_of_house/hosting/mod.rs* (older, still supported path)

If you use both for the same module, you'll get a compiler error. Using different styles for different modules in the same project is allowed, but might be confusing for people navigating your project.

The main downside to the style that uses files named *mod.rs* is that your project can end up with many files named *mod.rs*, which can get confusing when you have them open in your editor at the same time.

Moving each module's code to a separate file is now complete, and the module tree remains the same. The function calls in `eat_at_restaurant` will work without any modification, even though the definitions live in different files. This technique lets you move modules to new files as they grow in size.

Note that the `pub use crate::front_of_house::hosting` statement in *src/lib.rs* also hasn't changed, nor does `use` have any impact on what files are compiled as part of the crate. The `mod` keyword declares modules, and Rust looks in a file with the same name as the module for the code that goes into that module.

Summary

Rust lets you split a package into multiple crates and a crate into modules so you can refer to items defined in one module from another module. You can do this by specifying absolute or relative paths. These paths can be brought into scope with a `use` statement so you can use a shorter path for multiple uses of the item in that scope. Module code is private by default, but you can make definitions public by adding the `pub` keyword.

In the next chapter, we'll look at some collection data structures in the standard library that you can use in your neatly organized code.

Common Collections

Rust's standard library includes a number of very useful data structures called *collections*. Most other data types represent one specific value, but collections can contain multiple values. Unlike the built-in array and tuple types, the data these collections point to is stored on the heap, which means the amount of data does not need to be known at compile time and can grow or shrink as the program runs. Each kind of collection has different capabilities and costs, and choosing an appropriate one for your current situation is a skill you'll develop over time. In this chapter, we'll discuss three collections that are used very often in Rust programs:

- A *vector* allows you to store a variable number of values next to each other.
- A *string* is a collection of characters. We've mentioned the `String` type previously, but in this chapter we'll talk about it in depth.
- A *hash map* allows you to associate a value with a particular key. It's a particular implementation of the more general data structure called a *map*.

To learn about the other kinds of collections provided by the standard library, see the documentation.

We'll discuss how to create and update vectors, strings, and hash maps, as well as what makes each special.

Storing Lists of Values with Vectors

The first collection type we'll look at is `Vec<T>`, also known as a *vector*. Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. Vectors can only store values of the same type. They are useful when you have a list of items, such as the lines of text in a file or the prices of items in a shopping cart.

Creating a New Vector

To create a new empty vector, we call the `Vec::new` function, as shown in Listing 8-1.

```
    let v: Vec<i32> = Vec::new();
```

Listing 8-1: Creating a new, empty vector to hold values of type `i32`

Note that we added a type annotation here. Because we aren't inserting any values into this vector, Rust doesn't know what kind of elements we intend to store. This is an important point. Vectors are implemented using generics; we'll cover how to use generics with your own types in Chapter 10. For now, know that the `Vec<T>` type provided by the standard library can hold any type. When we create a vector to hold a specific type, we can specify the type within angle brackets. In Listing 8-1, we've told Rust that the `Vec<T>` in `v` will hold elements of the `i32` type.

More often, you'll create a `Vec<T>` with initial values and Rust will infer the type of value you want to store, so you rarely need to do this type annotation. Rust conveniently provides the `vec!` macro, which will create a new vector that holds the values you give it. Listing 8-2 creates a new `Vec<i32>` that holds the values `1`, `2`, and `3`. The integer type is `i32` because that's the default integer type, as we discussed in the "Data Types" section of Chapter 3.

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

Listing 8-2: Creating a new vector containing values

Because we've given initial `i32` values, Rust can infer that the type of `v` is `Vec<i32>`, and the type annotation isn't necessary. Next, we'll look at how to modify a vector.

Updating a Vector

To create a vector and then add elements to it, we can use the `push` method, as shown in Listing 8-3.

```
    let mut v = Vec::new();

    v.push(5);
    v.push(6);
    v.push(7);
    v.push(8);
```

Listing 8-3: Using the `push` method to add values to a vector

As with any variable, if we want to be able to change its value, we need to make it mutable using the `mut` keyword, as discussed in Chapter 3. The numbers we place inside are all of type `i32`, and Rust infers this from the data, so we don't need the `Vec<i32>` annotation.

Dropping a Vector Drops Its Elements

Like any other `struct`, a vector is freed when it goes out of scope, as annotated in Listing 8-4.

```
{
    let v = vec![1, 2, 3, 4];

    // do stuff with v
} // <- v goes out of scope and is freed here
```

Listing 8-4: Showing where the vector and its elements are dropped

When the vector gets dropped, all of its contents are also dropped, meaning those integers it holds will be cleaned up. This may seem like a straightforward point but it can get complicated when you start to introduce references to the elements of the vector. Let's tackle that next!

Reading Elements of Vectors

There are two ways to reference a value stored in a vector: via indexing or using the `get` method. In the following examples, we've annotated the types of the values that are returned from these functions for extra clarity.

Listing 8-5 shows both methods of accessing a value in a vector, with indexing syntax and the `get` method.

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {}", third);

match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

Listing 8-5: Using indexing syntax or the `get` method to access an item in a vector

Note two details here. First, we use the index value of `2` to get the third element because vectors are indexed by number, starting at zero. Second, we get the third element by either using `&` and `[]`, which gives us a reference, or using the `get` method with the index passed as an argument, which gives us an `Option<&T>`.

The reason Rust provides these two ways to reference an element is so you can choose how the program behaves when you try to use an index value outside the range of existing elements. As an example, let's see what happens when we have a vector of five elements and then we try to access an element at index 100 with each technique, as shown in Listing 8-6.

```
    let v = vec![1, 2, 3, 4, 5];

    let does_not_exist = &v[100];
    let does_not_exist = v.get(100);
```

Listing 8-6: Attempting to access the element at index 100 in a vector containing five elements

When we run this code, the first `[]` method will cause the program to panic because it references a nonexistent element. This method is best used when you want your program to crash if there's an attempt to access an element past the end of the vector.

When the `get` method is passed an index that is outside the vector, it returns `None` without panicking. You would use this method if accessing an element beyond the range of the vector may happen occasionally under normal circumstances. Your code will then have logic to handle having either `Some(&element)` or `None`, as discussed in Chapter 6. For example, the index could be coming from a person entering a number. If they accidentally enter a number that's too large and the program gets a `None` value, you could tell the user how many items are in the current vector and give them another chance to enter a valid value. That would be more user-friendly than crashing the program due to a typo!

When the program has a valid reference, the borrow checker enforces the ownership and borrowing rules (covered in Chapter 4) to ensure this reference and any other references to the contents of the

vector remain valid. Recall the rule that states you can't have mutable and immutable references in the same scope. That rule applies in Listing 8-7, where we hold an immutable reference to the first element in a vector and try to add an element to the end. This program won't work if we also try to refer to that element later in the function:

```
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    println!("The first element is: {}", first);
```

Listing 8-7: Attempting to add an element to a vector while holding a reference to an item

Compiling this code will result in this error:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let first = &v[0];
  |                  - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("The first element is: {}", first);
  |                                          ----- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` due to previous error
```

The code in Listing 8-7 might look like it should work: why should a reference to the first element care about changes at the end of the vector? This error is due to the way vectors work: because

vectors put the values next to each other in memory, adding a new element onto the end of the vector might require allocating new memory and copying the old elements to the new space, if there isn't enough room to put all the elements next to each other where the vector is currently stored. In that case, the reference to the first element would be pointing to deallocated memory. The borrowing rules prevent programs from ending up in that situation.

Note: For more on the implementation details of the `Vec<T>` type, see "The Rustonomicon".

Iterating over the Values in a Vector

To access each element in a vector in turn, we would iterate through all of the elements rather than use indices to access one at a time. Listing 8-8 shows how to use a `for` loop to get immutable references to each element in a vector of `i32` values and print them.

```
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{}", i);
    }
```

Listing 8-8: Printing each element in a vector by iterating over the elements using a `for` loop

We can also iterate over mutable references to each element in a mutable vector in order to make changes to all the elements. The `for` loop in Listing 8-9 will add `50` to each element.

```
    let mut v = vec![100, 32, 57];
    for i in &mut v {
        *i += 50;
    }
```

Listing 8-9: Iterating over mutable references to elements in a vector

To change the value that the mutable reference refers to, we have to use the `*` dereference operator to get to the value in `i` before we can use the `+=` operator. We'll talk more about the dereference operator in the "Following the Pointer to the Value with the Dereference Operator" section of Chapter 15.

Using an Enum to Store Multiple Types

Vectors can only store values that are the same type. This can be inconvenient; there are definitely use cases for needing to store a list of items of different types. Fortunately, the variants of an enum are defined under the same enum type, so when we need one type to represent elements of different types, we can define and use an enum!

For example, say we want to get values from a row in a spreadsheet in which some of the columns in the row contain integers, some floating-point numbers, and some strings. We can define an enum whose variants will hold the different value types, and all the enum variants will be considered the same type: that of the enum. Then we can create a vector to hold that enum and so, ultimately, holds different types. We've demonstrated this in Listing 8-10.

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Listing 8-10: Defining an `enum` to store values of different types in one vector

Rust needs to know what types will be in the vector at compile time so it knows exactly how much memory on the heap will be needed to store each element. We must also be explicit about what types are allowed in this vector. If Rust allowed a vector to hold any type, there would be a chance that one or more of the types would cause errors with the operations performed on the elements of the vector. Using an enum plus a `match` expression means that Rust will ensure at compile time that every possible case is handled, as discussed in Chapter 6.

If you don't know the exhaustive set of types a program will get at runtime to store in a vector, the enum technique won't work. Instead, you can use a trait object, which we'll cover in Chapter 17.

Now that we've discussed some of the most common ways to use vectors, be sure to review the API documentation for all the many useful methods defined on `Vec<T>` by the standard library. For example, in addition to `push`, a `pop` method removes and returns the last element. Let's move on to the next collection type: `String`!

Storing UTF-8 Encoded Text with Strings

We talked about strings in Chapter 4, but we'll look at them in more depth now. New Rustaceans commonly get stuck on strings for a combination of three reasons: Rust's propensity for exposing possible errors, strings being a more complicated data structure than many programmers give them credit for, and UTF-8. These factors combine in a way that can seem difficult when you're coming from other programming languages.

We discuss strings in the context of collections because strings are implemented as a collection of bytes, plus some methods to provide useful functionality when those bytes are interpreted as text. In this section, we'll talk about the operations on `String` that every collection type has, such as creating, updating, and reading. We'll also discuss the ways in which `String` is different from the other collections, namely how indexing into a `String` is complicated by the differences between how people and computers interpret `String` data.

What Is a String?

We'll first define what we mean by the term *string*. Rust has only one string type in the core language, which is the string slice `str` that is usually seen in its borrowed form `&str`. In Chapter 4, we talked about *string slices*, which are references to some UTF-8 encoded string data stored elsewhere. String literals, for example, are stored in the program's binary and are therefore string slices.

The `String` type, which is provided by Rust's standard library rather than coded into the core language, is a growable, mutable, owned, UTF-8 encoded string type. When Rustaceans refer to "strings" in Rust, they might be referring to either the `String` or the string slice `&str` types, not just one of those types. Although this section is largely about `String`, both types are used heavily in Rust's standard library, and both `String` and string slices are UTF-8 encoded.

Rust's standard library also includes a number of other string types, such as `OsString`, `OsStr`, `CString`, and `CStr`. Library crates can provide even more options for storing string data. See how those names all end in `String` or `Str`? They refer to owned and borrowed variants, just like the `String` and `str` types you've seen previously. These string types can store text in different encodings or be represented in memory in a different way, for example. We won't discuss these other string types in this chapter; see their API documentation for more about how to use them and when each is appropriate.

Creating a New String

Many of the same operations available with `Vec<T>` are available with `String` as well, starting with the `new` function to create a string, shown in Listing 8-11.

```
    let mut s = String::new();
```

Listing 8-11: Creating a new, empty `String`

This line creates a new empty string called `s`, which we can then load data into. Often, we'll have some initial data that we want to start the string with. For that, we use the `to_string` method, which is available on any type that implements the `Display` trait, as string literals do. Listing 8-12 shows two examples.

```
    let data = "initial contents";

    let s = data.to_string();

    // the method also works on a literal directly:
    let s = "initial contents".to_string();
```

Listing 8-12: Using the `to_string` method to create a `String` from a string literal

This code creates a string containing `initial contents`.

We can also use the function `String::from` to create a `String` from a string literal. The code in Listing 8-13 is equivalent to the code from Listing 8-12 that uses `to_string`.

```
    let s = String::from("initial contents");
```

Listing 8-13: Using the `String::from` function to create a `String` from a string literal

Because strings are used for so many things, we can use many different generic APIs for strings, providing us with a lot of options. Some of them can seem redundant, but they all have their place! In this case, `String::from` and `to_string` do the same thing, so which you choose is a matter of style and readability.

Remember that strings are UTF-8 encoded, so we can include any properly encoded data in them, as shown in Listing 8-14.

```
    let hello = String::from("السلام عليكم");
    let hello = String::from("Dobrý den");
    let hello = String::from("Hello");
    let hello = String::from("שלום");
    let hello = String::from("नमस्ते");
    let hello = String::from("こんにちは");
    let hello = String::from("안녕하세요");
    let hello = String::from("你好");
    let hello = String::from("Olá");
    let hello = String::from("Здравствуйте");
    let hello = String::from("Hola");
```

Listing 8-14: Storing greetings in different languages in strings

All of these are valid `String` values.

Updating a String

A `String` can grow in size and its contents can change, just like the contents of a `Vec<T>`, if you push more data into it. In addition, you can conveniently use the `+` operator or the `format!` macro to concatenate `String` values.

Appending to a String with `push_str` and `push`

We can grow a `String` by using the `push_str` method to append a string slice, as shown in Listing 8-15.

```
    let mut s = String::from("foo");
    s.push_str("bar");
```

Listing 8-15: Appending a string slice to a `String` using the `push_str` method

After these two lines, `s` will contain `foobar`. The `push_str` method takes a string slice because we don't necessarily want to take ownership of the parameter. For example, in the code in Listing 8-16, we want to be able to use `s2` after appending its contents to `s1`.

```
    let mut s1 = String::from("foo");
    let s2 = "bar";
    s1.push_str(s2);
    println!("s2 is {}", s2);
```

Listing 8-16: Using a string slice after appending its contents to a `String`

If the `push_str` method took ownership of `s2`, we wouldn't be able to print its value on the last line. However, this code works as we'd expect!

The `push` method takes a single character as a parameter and adds it to the `String`. Listing 8-17 adds the letter "l" to a `String` using the `push` method.

```
    let mut s = String::from("lo");
    s.push('l');
```

Listing 8-17: Adding one character to a `String` value using `push`

As a result, `s` will contain `lol`.

Concatenation with the `+` Operator or the `format!` Macro

Often, you'll want to combine two existing strings. One way to do so is to use the `+` operator, as shown in Listing 8-18.

```
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // note s1 has been moved here and can no longer be used
```

Listing 8-18: Using the `+` operator to combine two `String` values into a new `String` value

The string `s3` will contain `Hello, world!`. The reason `s1` is no longer valid after the addition, and the reason we used a reference to `s2`, has to do with the signature of the method that's called when we use the `+` operator. The `+` operator uses the `add` method, whose signature looks something like this:

```
fn add(self, s: &str) -> String {
```

In the standard library, you'll see `add` defined using generics. Here, we've substituted in concrete types for the generic ones, which is what happens when we call this method with `String` values. We'll discuss generics in Chapter 10. This signature gives us the clues we need to understand the tricky bits of the `+` operator.

First, `s2` has an `&`, meaning that we're adding a *reference* of the second string to the first string. This is because of the `s` parameter in the `add` function: we can only add a `&str` to a `String`; we can't add two `String` values together. But wait—the type of `&s2` is `&String`, not `&str`, as specified in the second parameter to `add`. So why does Listing 8-18 compile?

The reason we're able to use `&s2` in the call to `add` is that the compiler can *coerce* the `&String` argument into a `&str`. When we call the `add` method, Rust uses a *deref coercion*, which here turns `&s2` into `&s2[..]`. We'll discuss deref coercion in more depth in Chapter 15. Because `add` does not take ownership of the `s` parameter, `s2` will still be a valid `String` after this operation.

Second, we can see in the signature that `add` takes ownership of `self`, because `self` does *not* have an `&`. This means `s1` in Listing 8-18 will be moved into the `add` call and will no longer be valid after that. So although `let s3 = s1 + &s2;` looks like it will copy both strings and create a new one, this statement actually takes ownership of `s1`, appends a copy of the contents of `s2`, and then returns ownership of the result. In other words, it looks like it's making a lot of copies but isn't; the implementation is more efficient than copying.

If we need to concatenate multiple strings, the behavior of the `+` operator gets unwieldy:

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = s1 + "-" + &s2 + "-" + &s3;
```

At this point, `s` will be `tic-tac-toe`. With all of the `+` and `"` characters, it's difficult to see what's going on. For more complicated string combining, we can instead use the `format!` macro:

```
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");

    let s = format!("{}-{}-{}", s1, s2, s3);
```

This code also sets `s` to `tic-tac-toe`. The `format!` macro works like `println!`, but instead of printing the output to the screen, it returns a `String` with the contents. The version of the code using `format!` is much easier to read, and the code generated by the `format!` macro uses references so that this call doesn't take ownership of any of its parameters.

Indexing into Strings

In many other programming languages, accessing individual characters in a string by referencing them by index is a valid and common operation. However, if you try to access parts of a `String` using indexing syntax in Rust, you'll get an error. Consider the invalid code in Listing 8-19.

```
    let s1 = String::from("hello");
    let h = s1[0];
```

Listing 8-19: Attempting to use indexing syntax with a String

This code will result in the following error:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
3 |     let h = s1[0];
  |             ^^^^^ `String` cannot be indexed by `{integer}`
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` due to previous error
```

The error and the note tell the story: Rust strings don't support indexing. But why not? To answer that question, we need to discuss how Rust stores strings in memory.

Internal Representation

A `String` is a wrapper over a `Vec<u8>`. Let's look at some of our properly encoded UTF-8 example strings from Listing 8-14. First, this one:

```
    let hello = String::from("Hola");
```

In this case, `len` will be 4, which means the vector storing the string "Hola" is 4 bytes long. Each of these letters takes 1 byte when encoded in UTF-8. The following line, however, may surprise you. (Note that this string begins with the capital Cyrillic letter Ze, not the Arabic number 3.)

```
    let hello = String::from("Здравствуйте");
```

Asked how long the string is, you might say 12. In fact, Rust's answer is 24: that's the number of bytes it takes to encode "Здравствуйте" in UTF-8, because each Unicode scalar value in that string takes 2 bytes of storage. Therefore, an index into the string's bytes will not always correlate to a valid Unicode scalar value. To demonstrate, consider this invalid Rust code:

```
let hello = "Здравствуйте";
let answer = &hello[0];
```

You already know that `answer` will not be `3`, the first letter. When encoded in UTF-8, the first byte of `3` is `208` and the second is `151`, so it would seem that `answer` should in fact be `208`, but `208` is not a valid character on its own. Returning `208` is likely not what a user would want if they asked for the first letter of this string; however, that's the only data that Rust has at byte index 0. Users generally don't want the byte value returned, even if the string contains only Latin letters: if `&"hello"[0]` were valid code that returned the byte value, it would return `104`, not `h`.

The answer, then, is that to avoid returning an unexpected value and causing bugs that might not be discovered immediately, Rust doesn't compile this code at all and prevents misunderstandings early in the development process.

Bytes and Scalar Values and Grapheme Clusters! Oh My!

Another point about UTF-8 is that there are actually three relevant ways to look at strings from Rust's perspective: as bytes, scalar values, and grapheme clusters (the closest thing to what we would call *letters*).

If we look at the Hindi word "नमस्ते" written in the Devanagari script, it is stored as a vector of `u8` values that looks like this:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

That's 18 bytes and is how computers ultimately store this data. If we look at them as Unicode scalar values, which are what Rust's `char` type is, those bytes look like this:

```
['न', 'म', 'स', '्', 'त', 'े']
```

There are six `char` values here, but the fourth and sixth are not letters: they're diacritics that don't make sense on their own. Finally, if we look at them as grapheme clusters, we'd get what a person would call the four letters that make up the Hindi word:

```
["न", "म", "स्", "ते"]
```

Rust provides different ways of interpreting the raw string data that computers store so that each program can choose the interpretation it needs, no matter what human language the data is in.

A final reason Rust doesn't allow us to index into a `String` to get a character is that indexing operations are expected to always take constant time (O(1)). But it isn't possible to guarantee that performance with a `String`, because Rust would have to walk through the contents from the beginning to the index to determine how many valid characters there were.

Slicing Strings

Indexing into a string is often a bad idea because it's not clear what the return type of the string-indexing operation should be: a byte value, a character, a grapheme cluster, or a string slice. If you really need to use indices to create string slices, therefore, Rust asks you to be more specific.

Rather than indexing using `[]` with a single number, you can use `[]` with a range to create a string slice containing particular bytes:

```
let hello = "Здравствуйте";

let s = &hello[0..4];
```

Here, `s` will be a `&str` that contains the first 4 bytes of the string. Earlier, we mentioned that each of these characters was 2 bytes, which means `s` will be `Зд`.

If we were to try to slice only part of a character's bytes with something like `&hello[0..1]`, Rust would panic at runtime in the same way as if an invalid index were accessed in a vector:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside 'З' (bytes
0..2) of `Здравствуйте`', src/main.rs:4:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

You should use ranges to create string slices with caution, because doing so can crash your program.

Methods for Iterating Over Strings

The best way to operate on pieces of strings is to be explicit about whether you want characters or bytes. For individual Unicode scalar values, use the `chars` method. Calling `chars` on “नमस्ते” separates out and returns six values of type `char`, and you can iterate over the result to access each element:

```
for c in "नमस्ते".chars() {
    println!("{}", c);
}
```

This code will print the following:

```
न
म
स
ृ
त
े
```

Alternatively, the `bytes` method returns each raw byte, which might be appropriate for your domain:

```
for b in "नमस्ते".bytes() {
    println!("{}", b);
}
```

This code will print the 18 bytes that make up this `String`:

```
224
164
// --snip--
165
135
```

But be sure to remember that valid Unicode scalar values may be made up of more than 1 byte.

Getting grapheme clusters from strings is complex, so this functionality is not provided by the standard library. Crates are available on crates.io if this is the functionality you need.

Strings Are Not So Simple

To summarize, strings are complicated. Different programming languages make different choices about how to present this complexity to the programmer. Rust has chosen to make the correct handling of `String` data the default behavior for all Rust programs, which means programmers have to put more thought into handling UTF-8 data upfront. This trade-off exposes more of the complexity of strings than is apparent in other programming languages, but it prevents you from having to handle errors involving non-ASCII characters later in your development life cycle.

Let's switch to something a bit less complex: hash maps!

Storing Keys with Associated Values in Hash Maps

The last of our common collections is the *hash map*. The type `HashMap<K, V>` stores a mapping of keys of type `K` to values of type `V` using a *hashing function*, which determines how it places these keys and values into memory. Many programming languages support this kind of data structure, but they often use a different name, such as hash, map, object, hash table, dictionary, or associative array, just to name a few.

Hash maps are useful when you want to look up data not by using an index, as you can with vectors, but by using a key that can be of any type. For example, in a game, you could keep track of each team's score in a hash map in which each key is a team's name and the values are each team's score. Given a team name, you can retrieve its score.

We'll go over the basic API of hash maps in this section, but many more goodies are hiding in the functions defined on `HashMap<K, V>` by the standard library. As always, check the standard library documentation for more information.

Creating a New Hash Map

One way to create an empty hash map is using `new` and adding elements with `insert`. In Listing 8-20, we're keeping track of the scores of two teams whose names are *Blue* and *Yellow*. The Blue team starts with 10 points, and the Yellow team starts with 50.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

Listing 8-20: Creating a new hash map and inserting some keys and values

Note that we need to first `use` the `HashMap` from the collections portion of the standard library. Of our three common collections, this one is the least often used, so it's not included in the features brought into scope automatically in the prelude. Hash maps also have less support from the standard library; there's no built-in macro to construct them, for example.

Just like vectors, hash maps store their data on the heap. This `HashMap` has keys of type `String` and values of type `i32`. Like vectors, hash maps are homogeneous: all of the keys must have the same type, and all of the values must have the same type.

Another way of constructing a hash map is by using iterators and the `collect` method on a vector of tuples, where each tuple consists of a key and its value. We'll be going into more detail about iterators and their associated methods in the "Processing a Series of Items with Iterators" section of Chapter 13. The `collect` method gathers data into a number of collection types, including `HashMap`. For example, if we had the team names and initial scores in two separate vectors, we could use the `zip` method to create an iterator of tuples where "Blue" is paired with 10, and so forth. Then we could use the `collect` method to turn that iterator of tuples into a hash map, as shown in Listing 8-21.

```
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
    teams.into_iter().zip(initial_scores.into_iter()).collect();
```

Listing 8-21: Creating a hash map from a list of teams and a list of scores

The type annotation `HashMap<_, _>` is needed here because it's possible to `collect` into many different data structures and Rust doesn't know which you want unless you specify. For the parameters for the key and value types, however, we use underscores, and Rust can infer the types

that the hash map contains based on the types of the data in the vectors. In Listing 8-21, the key type will be `String` and the value type will be `i32`, just as in Listing 8-20.

Hash Maps and Ownership

For types that implement the `Copy` trait, like `i32`, the values are copied into the hash map. For owned values like `String`, the values will be moved and the hash map will be the owner of those values, as demonstrated in Listing 8-22.

```
    use std::collections::HashMap;

    let field_name = String::from("Favorite color");
    let field_value = String::from("Blue");

    let mut map = HashMap::new();
    map.insert(field_name, field_value);
    // field_name and field_value are invalid at this point, try using them and
    // see what compiler error you get!
```

Listing 8-22: Showing that keys and values are owned by the hash map once they're inserted

We aren't able to use the variables `field_name` and `field_value` after they've been moved into the hash map with the call to `insert`.

If we insert references to values into the hash map, the values won't be moved into the hash map. The values that the references point to must be valid for at least as long as the hash map is valid. We'll talk more about these issues in the "Validating References with Lifetimes" section in Chapter 10.

Accessing Values in a Hash Map

We can get a value out of the hash map by providing its key to the `get` method, as shown in Listing 8-23.

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

Listing 8-23: Accessing the score for the Blue team stored in the hash map

Here, `score` will have the value that's associated with the Blue team, and the result will be `Some(&10)`. The result is wrapped in `Some` because `get` returns an `Option<&V>`; if there's no value for that key in the hash map, `get` will return `None`. The program will need to handle the `Option` in one of the ways that we covered in Chapter 6.

We can iterate over each key/value pair in a hash map in a similar manner as we do with vectors, using a `for` loop:

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

This code will print each pair in an arbitrary order:

```
Yellow: 50
Blue: 10
```

Updating a Hash Map

Although the number of key and value pairs is growable, each key can only have one value associated with it at a time. When you want to change the data in a hash map, you have to decide how to handle the case when a key already has a value assigned. You could replace the old value with the new value, completely disregarding the old value. You could keep the old value and ignore the new value, only adding the new value if the key *doesn't* already have a value. Or you could combine the old value and the new value. Let's look at how to do each of these!

Overwriting a Value

If we insert a key and a value into a hash map and then insert that same key with a different value, the value associated with that key will be replaced. Even though the code in Listing 8-24 calls `insert` twice, the hash map will only contain one key/value pair because we're inserting the value for the Blue team's key both times.

```
    use std::collections::HashMap;

    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);

    println!("{:?}", scores);
```

Listing 8-24: Replacing a value stored with a particular key

This code will print `{"Blue": 25}`. The original value of `10` has been overwritten.

Only Inserting a Value If the Key Has No Value

It's common to check whether a particular key has a value and, if it doesn't, insert a value for it. Hash maps have a special API for this called `entry` that takes the key you want to check as a parameter. The return value of the `entry` method is an enum called `Entry` that represents a value that might or might not exist. Let's say we want to check whether the key for the Yellow team has a value associated with it. If it doesn't, we want to insert the value 50, and the same for the Blue team. Using the `entry` API, the code looks like Listing 8-25.

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

Listing 8-25: Using the `entry` method to only insert if the key does not already have a value

The `or_insert` method on `Entry` is defined to return a mutable reference to the value for the corresponding `Entry` key if that key exists, and if not, inserts the parameter as the new value for this key and returns a mutable reference to the new value. This technique is much cleaner than writing the logic ourselves and, in addition, plays more nicely with the borrow checker.

Running the code in Listing 8-25 will print `{"Yellow": 50, "Blue": 10}`. The first call to `entry` will insert the key for the Yellow team with the value 50 because the Yellow team doesn't have a value already. The second call to `entry` will not change the hash map because the Blue team already has the value 10.

Updating a Value Based on the Old Value

Another common use case for hash maps is to look up a key's value and then update it based on the old value. For instance, Listing 8-26 shows code that counts how many times each word appears in some text. We use a hash map with the words as keys and increment the value to keep track of how many times we've seen that word. If it's the first time we've seen a word, we'll first insert the value 0.

```
use std::collections::HashMap;

let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);
```

Listing 8-26: Counting occurrences of words using a hash map that stores words and counts

This code will print `{"world": 2, "hello": 1, "wonderful": 1}`. The `split_whitespace` method iterates over sub-slices, separated by whitespace, of the value in `text`. The `or_insert` method returns a mutable reference (`&mut V`) to the value for the specified key. Here we store that mutable reference in the `count` variable, so in order to assign to that value, we must first dereference `count` using the asterisk (`*`). The mutable reference goes out of scope at the end of the `for` loop, so all of these changes are safe and allowed by the borrowing rules.

Hashing Functions

By default, `HashMap` uses a hashing function called *SipHash* that can provide resistance to Denial of Service (DoS) attacks involving hash tables[1]. This is not the fastest hashing algorithm available, but the trade-off for better security that comes with the drop in performance is worth it. If you profile

your code and find that the default hash function is too slow for your purposes, you can switch to another function by specifying a different hasher. A *hasher* is a type that implements the `BuildHasher` trait. We'll talk about traits and how to implement them in Chapter 10. You don't necessarily have to implement your own hasher from scratch; crates.io has libraries shared by other Rust users that provide hashers implementing many common hashing algorithms.

[1] https://en.wikipedia.org/wiki/SipHash

Summary

Vectors, strings, and hash maps will provide a large amount of functionality necessary in programs when you need to store, access, and modify data. Here are some exercises you should now be equipped to solve:

- Given a list of integers, use a vector and return the median (when sorted, the value in the middle position) and mode (the value that occurs most often; a hash map will be helpful here) of the list.
- Convert strings to pig latin. The first consonant of each word is moved to the end of the word and "ay" is added, so "first" becomes "irst-fay." Words that start with a vowel have "hay" added to the end instead ("apple" becomes "apple-hay"). Keep in mind the details about UTF-8 encoding!
- Using a hash map and vectors, create a text interface to allow a user to add employee names to a department in a company. For example, "Add Sally to Engineering" or "Add Amir to Sales." Then let the user retrieve a list of all people in a department or all people in the company by department, sorted alphabetically.

The standard library API documentation describes methods that vectors, strings, and hash maps have that will be helpful for these exercises!

We're getting into more complex programs in which operations can fail, so, it's a perfect time to discuss error handling. We'll do that next!

Error Handling

Errors are a fact of life in software, so Rust has a number of features for handling situations in which something goes wrong. In many cases, Rust requires you to acknowledge the possibility of an error and take some action before your code will compile. This requirement makes your program more robust by ensuring that you'll discover errors and handle them appropriately before you've deployed your code to production!

Rust groups errors into two major categories: *recoverable* and *unrecoverable* errors. For a recoverable error, such as a *file not found* error, we most likely just want to report the problem to the user and retry the operation. Unrecoverable errors are always symptoms of bugs, like trying to access a location beyond the end of an array, and so we want to immediately stop the program.

Most languages don't distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Rust doesn't have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. This chapter covers calling `panic!` first and then talks about returning `Result<T, E>` values. Additionally, we'll explore considerations when deciding whether to try to recover from an error or to stop execution.

Unrecoverable Errors with `panic!`

Sometimes, bad things happen in your code, and there's nothing you can do about it. In these cases, Rust has the `panic!` macro. When the `panic!` macro executes, your program will print a failure message, unwind and clean up the stack, and then quit. We'll commonly invoke a panic when a bug of some kind has been detected and it's not clear how to handle the problem at the time we're writing our program.

Unwinding the Stack or Aborting in Response to a Panic

By default, when a panic occurs, the program starts *unwinding*, which means Rust walks back up the stack and cleans up the data from each function it encounters. However, this walking back and cleanup is a lot of work. Rust, therefore, allows you to choose the alternative of immediately *aborting*, which ends the program without cleaning up. Memory that the program was using will then need to be cleaned up by the operating system. If in your project you need to make the resulting binary as small as possible, you can switch from unwinding to aborting upon a panic by adding `panic = 'abort'` to the appropriate `[profile]` sections in your *Cargo.toml* file. For example, if you want to abort on panic in release mode, add this:

```
[profile.release]
panic = 'abort'
```

Let's try calling `panic!` in a simple program:

Filename: src/main.rs

```
fn main() {
    panic!("crash and burn");
}
```

When you run the program, you'll see something like this:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

The call to `panic!` causes the error message contained in the last two lines. The first line shows our panic message and the place in our source code where the panic occurred: *src/main.rs:2:5* indicates that it's the second line, fifth character of our *src/main.rs* file.

In this case, the line indicated is part of our code, and if we go to that line, we see the `panic!` macro call. In other cases, the `panic!` call might be in code that our code calls, and the filename and line number reported by the error message will be someone else's code where the `panic!` macro is called, not the line of our code that eventually led to the `panic!` call. We can use the backtrace of the functions the `panic!` call came from to figure out the part of our code that is causing the problem. We'll discuss backtraces in more detail next.

Using a `panic!` Backtrace

Let's look at another example to see what it's like when a `panic!` call comes from a library because of a bug in our code instead of from our code calling the macro directly. Listing 9-1 has some code that attempts to access an index in a vector beyond the range of valid indexes.

Filename: src/main.rs

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

Listing 9-1: Attempting to access an element beyond the end of a vector, which will cause a call to `panic!`

Here, we're attempting to access the 100th element of our vector (which is at index 99 because indexing starts at zero), but the vector has only 3 elements. In this situation, Rust will panic. Using `[]` is supposed to return an element, but if you pass an invalid index, there's no element that Rust could return here that would be correct.

In C, attempting to read beyond the end of a data structure is undefined behavior. You might get whatever is at the location in memory that would correspond to that element in the data structure, even though the memory doesn't belong to that structure. This is called a *buffer overread* and can lead to security vulnerabilities if an attacker is able to manipulate the index in such a way as to read data they shouldn't be allowed to that is stored after the data structure.

To protect your program from this sort of vulnerability, if you try to read an element at an index that doesn't exist, Rust will stop execution and refuse to continue. Let's try it and see:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

This error points at line 4 of our `main.rs` where we attempt to access index 99. The next note line tells us that we can set the `RUST_BACKTRACE` environment variable to get a backtrace of exactly what happened to cause the error. A *backtrace* is a list of all the functions that have been called to get to

this point. Backtraces in Rust work as they do in other languages: the key to reading the backtrace is to start from the top and read until you see files you wrote. That's the spot where the problem originated. The lines above that spot are code that your code has called; the lines below are code that called your code. These before-and-after lines might include core Rust code, standard library code, or crates that you're using. Let's try getting a backtrace by setting the `RUST_BACKTRACE` environment variable to any value except 0. Listing 9-2 shows output similar to what you'll see.

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99',
src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/std/src/panicking.rs:483
   1: core::panicking::panic_fmt
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/panicking.rs:85
   2: core::panicking::panic_bounds_check
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/panicking.rs:62
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/slice/index.rs:255
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/slice/index.rs:15
   5: <alloc::vec::Vec<T> as core::ops::index::Index<I>>::index
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/alloc/src/vec.rs:1982
   6: panic::main
             at ./src/main.rs:4
   7: core::ops::function::FnOnce::call_once
             at
/rustc/7eac88abb2e57e752f3302f02be5f3ce3d7adfb4/library/core/src/ops/function.rs:227
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Listing 9-2: The backtrace generated by a call to `panic!` displayed when the environment variable `RUST_BACKTRACE` is set

That's a lot of output! The exact output you see might be different depending on your operating system and Rust version. In order to get backtraces with this information, debug symbols must be enabled. Debug symbols are enabled by default when using `cargo build` or `cargo run` without the `--release` flag, as we have here.

In the output in Listing 9-2, line 6 of the backtrace points to the line in our project that's causing the problem: line 4 of *src/main.rs*. If we don't want our program to panic, we should start our investigation at the location pointed to by the first line mentioning a file we wrote. In Listing 9-1, where we deliberately wrote code that would panic, the way to fix the panic is to not request an element beyond the range of the vector indexes. When your code panics in the future, you'll need to figure out what action the code is taking with what values to cause the panic and what the code should do instead.

We'll come back to `panic!` and when we should and should not use `panic!` to handle error conditions in the "To `panic!` or Not to `panic!`" section later in this chapter. Next, we'll look at how to recover from an error using `Result`.

Recoverable Errors with `Result`

Most errors aren't serious enough to require the program to stop entirely. Sometimes, when a function fails, it's for a reason that you can easily interpret and respond to. For example, if you try to open a file and that operation fails because the file doesn't exist, you might want to create the file instead of terminating the process.

Recall from "Handling Potential Failure with the `Result` Type" in Chapter 2 that the `Result` enum is defined as having two variants, `Ok` and `Err`, as follows:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `T` and `E` are generic type parameters: we'll discuss generics in more detail in Chapter 10. What you need to know right now is that `T` represents the type of the value that will be returned in a success case within the `Ok` variant, and `E` represents the type of the error that will be returned in a failure case within the `Err` variant. Because `Result` has these generic type parameters, we can use the `Result` type and the functions defined on it in many different situations where the successful value and error value we want to return may differ.

Let's call a function that returns a `Result` value because the function could fail. In Listing 9-3 we try to open a file.

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");
}
```

Listing 9-3: Opening a file

How do we know `File::open` returns a `Result`? We could look at the standard library API documentation, or we could ask the compiler! If we give `f` a type annotation that we know is *not* the return type of the function and then try to compile the code, the compiler will tell us that the types don't match. The error message will then tell us what the type of `f` *is*. Let's try it! We know that the return type of `File::open` isn't of type `u32`, so let's change the `let f` statement to this:

```
    let f: u32 = File::open("hello.txt");
```

Attempting to compile now gives us the following output:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0308]: mismatched types
 --> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |            ---   ^^^^^^^^^^^^^^^^^^^^^^^ expected `u32`, found enum `Result`
  |            |
  |            expected due to this
  |
  = note: expected type `u32`
             found enum `Result<File, std::io::Error>`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `error-handling` due to previous error
```

This tells us the return type of the `File::open` function is a `Result<T, E>`. The generic parameter `T` has been filled in here with the type of the success value, `std::fs::File`, which is a file handle. The type of `E` used in the error value is `std::io::Error`.

This return type means the call to `File::open` might succeed and return a file handle that we can read from or write to. The function call also might fail: for example, the file might not exist, or we

might not have permission to access the file. The `File::open` function needs to have a way to tell us whether it succeeded or failed and at the same time give us either the file handle or error information. This information is exactly what the `Result` enum conveys.

In the case where `File::open` succeeds, the value in the variable `f` will be an instance of `Ok` that contains a file handle. In the case where it fails, the value in `f` will be an instance of `Err` that contains more information about the kind of error that happened.

We need to add to the code in Listing 9-3 to take different actions depending on the value `File::open` returns. Listing 9-4 shows one way to handle the `Result` using a basic tool, the `match` expression that we discussed in Chapter 6.

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Listing 9-4: Using a `match` expression to handle the `Result` variants that might be returned

Note that, like the `Option` enum, the `Result` enum and its variants have been brought into scope by the prelude, so we don't need to specify `Result::` before the `Ok` and `Err` variants in the `match` arms.

When the result is `Ok`, this code will return the inner `file` value out of the `Ok` variant, and we then assign that file handle value to the variable `f`. After the `match`, we can use the file handle for reading or writing.

The other arm of the `match` handles the case where we get an `Err` value from `File::open`. In this example, we've chosen to call the `panic!` macro. If there's no file named *hello.txt* in our current directory and we run this code, we'll see the following output from the `panic!` macro:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at 'Problem opening the file: Os { code: 2, kind: NotFound,
message: "No such file or directory" }', src/main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As usual, this output tells us exactly what has gone wrong.

Matching on Different Errors

The code in Listing 9-4 will `panic!` no matter why `File::open` failed. However, we want to take different actions for different failure reasons: if `File::open` failed because the file doesn't exist, we want to create the file and return the handle to the new file. If `File::open` failed for any other reason—for example, because we didn't have permission to open the file—we still want the code to `panic!` in the same way as it did in Listing 9-4. For this we add an inner `match` expression, shown in Listing 9-5.

Filename: src/main.rs

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error)
            }
        },
    };
}
```

Listing 9-5: Handling different kinds of errors in different ways

The type of the value that `File::open` returns inside the `Err` variant is `io::Error`, which is a struct provided by the standard library. This struct has a method `kind` that we can call to get an `io::ErrorKind` value. The enum `io::ErrorKind` is provided by the standard library and has variants representing the different kinds of errors that might result from an `io` operation. The variant we want to use is `ErrorKind::NotFound`, which indicates the file we're trying to open doesn't exist yet. So we match on `f`, but we also have an inner match on `error.kind()`.

The condition we want to check in the inner match is whether the value returned by `error.kind()` is the `NotFound` variant of the `ErrorKind` enum. If it is, we try to create the file with `File::create`. However, because `File::create` could also fail, we need a second arm in the inner `match` expression. When the file can't be created, a different error message is printed. The second arm of the outer `match` stays the same, so the program panics on any error besides the missing file error.

Alternatives to Using `match` with `Result<T, E>`

That's a lot of `match`! The `match` expression is very useful but also very much a primitive. In Chapter 13, you'll learn about closures, which are used with many of the methods defined on `Result<T, E>`. These methods can be more concise than using `match` when handling `Result<T, E>` values in your code.

For example, here's another way to write the same logic as shown in Listing 9-5 but using closures and the `unwrap_or_else` method:

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

Although this code has the same behavior as Listing 9-5, it doesn't contain any `match` expressions and is cleaner to read. Come back to this example after you've read Chapter 13, and look up the `unwrap_or_else` method in the standard library documentation. Many more of these methods can clean up huge nested `match` expressions when you're dealing with errors.

Shortcuts for Panic on Error: `unwrap` and `expect`

Using `match` works well enough, but it can be a bit verbose and doesn't always communicate intent well. The `Result<T, E>` type has many helper methods defined on it to do various, more specific tasks. The `unwrap` method is a shortcut method implemented just like the `match` expression we wrote in Listing 9-4. If the `Result` value is the `Ok` variant, `unwrap` will return the value inside the `Ok`. If the `Result` is the `Err` variant, `unwrap` will call the `panic!` macro for us. Here is an example of `unwrap` in action:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

If we run this code without a *hello.txt* file, we'll see an error message from the `panic!` call that the `unwrap` method makes:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Error {
repr: Os { code: 2, message: "No such file or directory" } }',
src/libcore/result.rs:906:4
```

Similarly, the `expect` method lets us also choose the `panic!` error message. Using `expect` instead of `unwrap` and providing good error messages can convey your intent and make tracking down the source of a panic easier. The syntax of `expect` looks like this:

Filename: src/main.rs

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
```

We use `expect` in the same way as `unwrap`: to return the file handle or call the `panic!` macro. The error message used by `expect` in its call to `panic!` will be the parameter that we pass to `expect`, rather than the default `panic!` message that `unwrap` uses. Here's what it looks like:

```
thread 'main' panicked at 'Failed to open hello.txt: Error { repr: Os { code:
2, message: "No such file or directory" } }', src/libcore/result.rs:906:4
```

Because this error message starts with the text we specified, `Failed to open hello.txt`, it will be easier to find where in the code this error message is coming from. If we use `unwrap` in multiple places, it can take more time to figure out exactly which `unwrap` is causing the panic because all `unwrap` calls that panic print the same message.

Propagating Errors

When a function's implementation calls something that might fail, instead of handling the error within the function itself, you can return the error to the calling code so that it can decide what to do. This is known as *propagating* the error and gives more control to the calling code, where there might be more information or logic that dictates how the error should be handled than what you have available in the context of your code.

For example, Listing 9-6 shows a function that reads a username from a file. If the file doesn't exist or can't be read, this function will return those errors to the code that called the function.

Filename: src/main.rs

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

Listing 9-6: A function that returns errors to the calling code using `match`

This function can be written in a much shorter way, but we're going to start by doing a lot of it manually in order to explore error handling; at the end, we'll show the shorter way. Let's look at the return type of the function first: `Result<String, io::Error>`. This means the function is returning a value of the type `Result<T, E>` where the generic parameter `T` has been filled in with the concrete type `String`, and the generic type `E` has been filled in with the concrete type `io::Error`. If this function succeeds without any problems, the code that calls this function will receive an `Ok` value that holds a `String`—the username that this function read from the file. If this function encounters any problems, the calling code will receive an `Err` value that holds an instance of `io::Error` that contains more information about what the problems were. We chose `io::Error` as the return type of this function because that happens to be the type of the error value returned from both of the operations we're calling in this function's body that might fail: the `File::open` function and the `read_to_string` method.

The body of the function starts by calling the `File::open` function. Then we handle the `Result` value with a `match` similar to the `match` in Listing 9-4. If `File::open` succeeds, the file handle in the pattern variable `file` becomes the value in the mutable variable `f` and the function continues. In the `Err` case, instead of calling `panic!`, we use the `return` keyword to return early out of the function entirely and pass the error value from `File::open`, now in the pattern variable `e`, back to the calling code as this function's error value.

So if we have a file handle in `f`, the function then creates a new `String` in variable `s` and calls the `read_to_string` method on the file handle in `f` to read the contents of the file into `s`. The `read_to_string` method also returns a `Result` because it might fail, even though `File::open` succeeded. So we need another `match` to handle that `Result`: if `read_to_string` succeeds, then our function has succeeded, and we return the username from the file that's now in `s` wrapped in an `Ok`. If `read_to_string` fails, we return the error value in the same way that we returned the error value in the `match` that handled the return value of `File::open`. However, we don't need to explicitly say `return`, because this is the last expression in the function.

The code that calls this code will then handle getting either an `Ok` value that contains a username or an `Err` value that contains an `io::Error`. It's up to the calling code to decide what to do with those values. If the calling code gets an `Err` value, it could call `panic!` and crash the program, use a default username, or look up the username from somewhere other than a file, for example. We don't have enough information on what the calling code is actually trying to do, so we propagate all the success or error information upward for it to handle appropriately.

This pattern of propagating errors is so common in Rust that Rust provides the question mark operator `?` to make this easier.

A Shortcut for Propagating Errors: the ? Operator

Listing 9-7 shows an implementation of `read_username_from_file` that has the same functionality as in Listing 9-6, but this implementation uses the `?` operator.

Filename: src/main.rs

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Listing 9-7: A function that returns errors to the calling code using the `?` operator

The `?` placed after a `Result` value is defined to work in almost the same way as the `match` expressions we defined to handle the `Result` values in Listing 9-6. If the value of the `Result` is an `Ok`, the value inside the `Ok` will get returned from this expression, and the program will continue. If the value is an `Err`, the `Err` will be returned from the whole function as if we had used the `return` keyword so the error value gets propagated to the calling code.

There is a difference between what the `match` expression from Listing 9-6 does and what the `?` operator does: error values that have the `?` operator called on them go through the `from` function, defined in the `From` trait in the standard library, which is used to convert errors from one type into another. When the `?` operator calls the `from` function, the error type received is converted into the error type defined in the return type of the current function. This is useful when a function returns one error type to represent all the ways a function might fail, even if parts might fail for many different reasons. As long as there's an `impl From<OtherError> for ReturnedError` to define the conversion in the trait's `from` function, the `?` operator takes care of calling the `from` function automatically.

In the context of Listing 9-7, the `?` at the end of the `File::open` call will return the value inside an `Ok` to the variable `f`. If an error occurs, the `?` operator will return early out of the whole function and give any `Err` value to the calling code. The same thing applies to the `?` at the end of the `read_to_string` call.

The `?` operator eliminates a lot of boilerplate and makes this function's implementation simpler. We could even shorten this code further by chaining method calls immediately after the `?`, as shown in Listing 9-8.

Filename: src/main.rs

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();

    File::open("hello.txt")?.read_to_string(&mut s)?;

    Ok(s)
}
```

Listing 9-8: Chaining method calls after the `?` operator

We've moved the creation of the new `String` in `s` to the beginning of the function; that part hasn't changed. Instead of creating a variable `f`, we've chained the call to `read_to_string` directly onto the result of `File::open("hello.txt")?`. We still have a `?` at the end of the `read_to_string` call, and we still return an `Ok` value containing the username in `s` when both `File::open` and `read_to_string` succeed rather than returning errors. The functionality is again the same as in Listing 9-6 and Listing 9-7; this is just a different, more ergonomic way to write it.

Listing 9-9 shows a way to make this even shorter using `fs::read_to_string`.

Filename: src/main.rs

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

Listing 9-9: Using `fs::read_to_string` instead of opening and then reading the file

Reading a file into a string is a fairly common operation, so the standard library provides the convenient `fs::read_to_string` function that opens the file, creates a new `String`, reads the contents of the file, puts the contents into that `String`, and returns it. Of course, using `fs::read_to_string` doesn't give us the opportunity to explain all the error handling, so we did it the longer way first.

Where The `?` Operator Can Be Used

The `?` operator can only be used in functions whose return type is compatible with the value the `?` is used on. This is because the `?` operator is defined to perform an early return of a value out of the function, in the same manner as the `match` expression we defined in Listing 9-6. In Listing 9-6, the `match` was using a `Result` value, and the early return arm returned an `Err(e)` value. The return type of the function has to be a `Result` so that it's compatible with this `return`.

In Listing 9-10, let's look at the error we'll get if we use the `?` operator in a `main` function with a return type incompatible with the type of the value we use `?` on:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt")?;
}
```

Listing 9-10: Attempting to use the `?` in the `main` function that returns `()` won't compile

This code opens a file, which might fail. The `?` operator follows the `Result` value returned by `File::open`, but this `main` function has the return type of `()`, not `Result`. When we compile this code, we get the following error message:

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `Result` or
`Option` (or another type that implements `FromResidual`)
 --> src/main.rs:4:36
  |
3 | / fn main() {
4 | |     let f = File::open("hello.txt")?;
  | |                                    ^ cannot use the `?` operator in a function
that returns `()`
5 | | }
  | |_- this function should return `Result` or `Option` to accept `?`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not
implemented for `()`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `error-handling` due to previous error
```

This error points out that we're only allowed to use the `?` operator in a function that returns `Result`, `Option`, or another type that implements `FromResidual`. To fix the error, you have two choices. One choice is to change the return type of your function to be compatible with the value you're using the `?` operator on as long as you have no restrictions preventing that. The other

technique is to use a `match` or one of the `Result<T, E>` methods to handle the `Result<T, E>` in whatever way is appropriate.

The error message also mentioned that `?` can be used with `Option<T>` values as well. As with using `?` on `Result`, you can only use `?` on `Option` in a function that returns an `Option`. The behavior of the `?` operator when called on an `Option<T>` is similar to its behavior when called on a `Result<T, E>`: if the value is `None`, the `None` will be returned early from the function at that point. If the value is `Some`, the value inside the `Some` is the resulting value of the expression and the function continues. Listing 9-11 has an example of a function that finds the last character of the first line in the given text:

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

Listing 9-11: Using the `?` operator on an `Option<T>` value

This function returns `Option<char>` because it's possible that there is a character there, but it's also possible that there isn't. This code takes the `text` string slice argument and calls the `lines` method on it, which returns an iterator over the lines in the string. Because this function wants to examine the first line, it calls `next` on the iterator to get the first value from the iterator. If `text` is the empty string, this call to `next` will return `None`, in which case we use `?` to stop and return `None` from `last_char_of_first_line`. If `text` is not the empty string, `next` will return a `Some` value containing a string slice of the first line in `text`.

The `?` extracts the string slice, and we can call `chars` on that string slice to get an iterator of its characters. We're interested in the last character in this first line, so we call `last` to return the last item in the iterator. This is an `Option` because it's possible that the first line is the empty string, for example if `text` starts with a blank line but has characters on other lines, as in `"\nhi"`. However, if there is a last character on the first line, it will be returned in the `Some` variant. The `?` operator in the middle gives us a concise way to express this logic, allowing us to implement the function in one

line. If we couldn't use the `?` operator on `Option`, we'd have to implement this logic using more method calls or a `match` expression.

Note that you can use the `?` operator on a `Result` in a function that returns `Result`, and you can use the `?` operator on an `Option` in a function that returns `Option`, but you can't mix and match. The `?` operator won't automatically convert a `Result` to an `Option` or vice versa; in those cases, you can use methods like the `ok` method on `Result` or the `ok_or` method on `Option` to do the conversion explicitly.

So far, all the `main` functions we've used return `()`. The `main` function is special because it's the entry and exit point of executable programs, and there are restrictions on what its return type can be for the programs to behave as expected.

Luckily, `main` can also return a `Result<(), E>`. Listing 9-12 has the code from Listing 9-10 but we've changed the return type of `main` to be `Result<(), Box<dyn Error>>` and added a return value `Ok(())` to the end. This code will now compile:

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let f = File::open("hello.txt")?;

    Ok(())
}
```

Listing 9-12: Changing `main` to return `Result<(), E>` allows the use of the `?` operator on `Result` values

The `Box<dyn Error>` type is a *trait object*, which we'll talk about in the "Using Trait Objects that Allow for Values of Different Types" section in Chapter 17. For now, you can read `Box<dyn Error>` to mean "any kind of error." Using `?` on a `Result` value in a `main` function with the error type `Box<dyn Error>` is allowed, because it allows any `Err` value to be returned early.

When a `main` function returns a `Result<(), E>`, the executable will exit with a value of `0` if `main` returns `Ok(())` and will exit with a nonzero value if `main` returns an `Err` value. Executables written in C return integers when they exit: programs that exit successfully return the integer `0`, and programs that error return some integer other than `0`. Rust also returns integers from executables to be compatible with this convention.

The `main` function may return any types that implement the `std::process::Termination` trait. As of this writing, the `Termination` trait is an unstable feature only available in Nightly Rust, so you can't yet implement it for your own types in Stable Rust, but you might be able to someday!

Now that we've discussed the details of calling `panic!` or returning `Result`, let's return to the topic of how to decide which is appropriate to use in which cases.

To `panic!` or Not to `panic!`

So how do you decide when you should call `panic!` and when you should return `Result`? When code panics, there's no way to recover. You could call `panic!` for any error situation, whether there's a possible way to recover or not, but then you're making the decision that a situation is unrecoverable on behalf of the calling code. When you choose to return a `Result` value, you give the calling code options. The calling code could choose to attempt to recover in a way that's appropriate for its situation, or it could decide that an `Err` value in this case is unrecoverable, so it can call `panic!` and turn your recoverable error into an unrecoverable one. Therefore, returning `Result` is a good default choice when you're defining a function that might fail.

In situations such as examples, prototype code, and tests, it's more appropriate to write code that panics instead of returning a `Result`. Let's explore why, then discuss situations in which the compiler can't tell that failure is impossible, but you as a human can. The chapter will conclude with some general guidelines on how to decide whether to panic in library code.

Examples, Prototype Code, and Tests

When you're writing an example to illustrate some concept, also including robust error-handling code can make the example less clear. In examples, it's understood that a call to a method like `unwrap` that could panic is meant as a placeholder for the way you'd want your application to handle errors, which can differ based on what the rest of your code is doing.

Similarly, the `unwrap` and `expect` methods are very handy when prototyping, before you're ready to decide how to handle errors. They leave clear markers in your code for when you're ready to make your program more robust.

If a method call fails in a test, you'd want the whole test to fail, even if that method isn't the functionality under test. Because `panic!` is how a test is marked as a failure, calling `unwrap` or `expect` is exactly what should happen.

Cases in Which You Have More Information Than the Compiler

It would also be appropriate to call `unwrap` when you have some other logic that ensures the `Result` will have an `Ok` value, but the logic isn't something the compiler understands. You'll still have a `Result` value that you need to handle: whatever operation you're calling still has the possibility of failing in general, even though it's logically impossible in your particular situation. If you can ensure by manually inspecting the code that you'll never have an `Err` variant, it's perfectly acceptable to call `unwrap`. Here's an example:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1".parse().unwrap();
```

We're creating an `IpAddr` instance by parsing a hardcoded string. We can see that `127.0.0.1` is a valid IP address, so it's acceptable to use `unwrap` here. However, having a hardcoded, valid string doesn't change the return type of the `parse` method: we still get a `Result` value, and the compiler will still make us handle the `Result` as if the `Err` variant is a possibility because the compiler isn't smart enough to see that this string is always a valid IP address. If the IP address string came from a user rather than being hardcoded into the program and therefore *did* have a possibility of failure, we'd definitely want to handle the `Result` in a more robust way instead.

Guidelines for Error Handling

It's advisable to have your code panic when it's possible that your code could end up in a bad state. In this context, a *bad state* is when some assumption, guarantee, contract, or invariant has been broken, such as when invalid values, contradictory values, or missing values are passed to your code —plus one or more of the following:

- The bad state is something that is unexpected, as opposed to something that will likely happen occasionally, like a user entering data in the wrong format.

- Your code after this point needs to rely on not being in this bad state, rather than checking for the problem at every step.
- There's not a good way to encode this information in the types you use. We'll work through an example of what we mean in the "Encoding States and Behavior as Types" section of Chapter 17.

If someone calls your code and passes in values that don't make sense, the best choice might be to call `panic!` and alert the person using your library to the bug in their code so they can fix it during development. Similarly, `panic!` is often appropriate if you're calling external code that is out of your control and it returns an invalid state that you have no way of fixing.

However, when failure is expected, it's more appropriate to return a `Result` than to make a `panic!` call. Examples include a parser being given malformed data or an HTTP request returning a status that indicates you have hit a rate limit. In these cases, returning a `Result` indicates that failure is an expected possibility that the calling code must decide how to handle.

When your code performs operations on values, your code should verify the values are valid first and panic if the values aren't valid. This is mostly for safety reasons: attempting to operate on invalid data can expose your code to vulnerabilities. This is the main reason the standard library will call `panic!` if you attempt an out-of-bounds memory access: trying to access memory that doesn't belong to the current data structure is a common security problem. Functions often have *contracts*: their behavior is only guaranteed if the inputs meet particular requirements. Panicking when the contract is violated makes sense because a contract violation always indicates a caller-side bug and it's not a kind of error you want the calling code to have to explicitly handle. In fact, there's no reasonable way for calling code to recover; the calling *programmers* need to fix the code. Contracts for a function, especially when a violation will cause a panic, should be explained in the API documentation for the function.

However, having lots of error checks in all of your functions would be verbose and annoying. Fortunately, you can use Rust's type system (and thus the type checking done by the compiler) to do many of the checks for you. If your function has a particular type as a parameter, you can proceed

with your code's logic knowing that the compiler has already ensured you have a valid value. For example, if you have a type rather than an `Option`, your program expects to have *something* rather than *nothing*. Your code then doesn't have to handle two cases for the `Some` and `None` variants: it will only have one case for definitely having a value. Code trying to pass nothing to your function won't even compile, so your function doesn't have to check for that case at runtime. Another example is using an unsigned integer type such as `u32`, which ensures the parameter is never negative.

Creating Custom Types for Validation

Let's take the idea of using Rust's type system to ensure we have a valid value one step further and look at creating a custom type for validation. Recall the guessing game in Chapter 2 in which our code asked the user to guess a number between 1 and 100. We never validated that the user's guess was between those numbers before checking it against our secret number; we only validated that the guess was positive. In this case, the consequences were not very dire: our output of "Too high" or "Too low" would still be correct. But it would be a useful enhancement to guide the user toward valid guesses and have different behavior when a user guesses a number that's out of range versus when a user types, for example, letters instead.

One way to do this would be to parse the guess as an `i32` instead of only a `u32` to allow potentially negative numbers, and then add a check for the number being in range, like so:

```
    loop {
        // --snip--

        let guess: i32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        if guess < 1 || guess > 100 {
            println!("The secret number will be between 1 and 100.");
            continue;
        }

        match guess.cmp(&secret_number) {
            // --snip--
    }
```

The `if` expression checks whether our value is out of range, tells the user about the problem, and calls `continue` to start the next iteration of the loop and ask for another guess. After the `if` expression, we can proceed with the comparisons between `guess` and the secret number knowing that `guess` is between 1 and 100.

However, this is not an ideal solution: if it was absolutely critical that the program only operated on values between 1 and 100, and it had many functions with this requirement, having a check like this in every function would be tedious (and might impact performance).

Instead, we can make a new type and put the validations in a function to create an instance of the type rather than repeating the validations everywhere. That way, it's safe for functions to use the new type in their signatures and confidently use the values they receive. Listing 9-13 shows one way to define a `Guess` type that will only create an instance of `Guess` if the `new` function receives a value between 1 and 100.

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Listing 9-13: A `Guess` type that will only continue with values between 1 and 100

First, we define a struct named `Guess` that has a field named `value` that holds an `i32`. This is where the number will be stored.

Then we implement an associated function named `new` on `Guess` that creates instances of `Guess` values. The `new` function is defined to have one parameter named `value` of type `i32` and to return a `Guess`. The code in the body of the `new` function tests `value` to make sure it's between 1 and 100. If `value` doesn't pass this test, we make a `panic!` call, which will alert the programmer who is writing the calling code that they have a bug they need to fix, because creating a `Guess` with a `value` outside this range would violate the contract that `Guess::new` is relying on. The conditions in which `Guess::new` might panic should be discussed in its public-facing API documentation; we'll cover documentation conventions indicating the possibility of a `panic!` in the API documentation that you create in Chapter 14. If `value` does pass the test, we create a new `Guess` with its `value` field set to the `value` parameter and return the `Guess`.

Next, we implement a method named `value` that borrows `self`, doesn't have any other parameters, and returns an `i32`. This kind of method is sometimes called a *getter*, because its purpose is to get some data from its fields and return it. This public method is necessary because the `value` field of the `Guess` struct is private. It's important that the `value` field be private so code using the `Guess` struct is not allowed to set `value` directly: code outside the module *must* use the `Guess::new` function to create an instance of `Guess`, thereby ensuring there's no way for a `Guess` to have a `value` that hasn't been checked by the conditions in the `Guess::new` function.

A function that has a parameter or returns only numbers between 1 and 100 could then declare in its signature that it takes or returns a `Guess` rather than an `i32` and wouldn't need to do any additional checks in its body.

Summary

Rust's error handling features are designed to help you write more robust code. The `panic!` macro signals that your program is in a state it can't handle and lets you tell the process to stop instead of trying to proceed with invalid or incorrect values. The `Result` enum uses Rust's type system to indicate that operations might fail in a way that your code could recover from. You can use `Result` to tell code that calls your code that it needs to handle potential success or failure as well. Using `panic!` and `Result` in the appropriate situations will make your code more reliable in the face of inevitable problems.

Now that you've seen useful ways that the standard library uses generics with the `Option` and `Result` enums, we'll talk about how generics work and how you can use them in your code.

Generic Types, Traits, and Lifetimes

Every programming language has tools for effectively handling the duplication of concepts. In Rust, one such tool is *generics*: abstract stand-ins for concrete types or other properties. We can express the behavior of generics or how they relate to other generics without knowing what will be in their place when compiling and running the code.

Functions can take parameters of some generic type, instead of a concrete type like `i32` or `String`, in the same way a function takes parameters with unknown values to run the same code on multiple concrete values. In fact, we've already used generics in Chapter 6 with `Option<T>`, Chapter 8 with `Vec<T>` and `HashMap<K, V>`, and Chapter 9 with `Result<T, E>`. In this chapter, you'll explore how to define your own types, functions, and methods with generics!

First, we'll review how to extract a function to reduce code duplication. We'll then use the same technique to make a generic function from two functions that differ only in the types of their parameters. We'll also explain how to use generic types in struct and enum definitions.

Then you'll learn how to use *traits* to define behavior in a generic way. You can combine traits with generic types to constrain a generic type to accept only those types that have a particular behavior, as opposed to just any type.

Finally, we'll discuss *lifetimes*: a variety of generics that give the compiler information about how references relate to each other. Lifetimes allow us to give the compiler enough information about borrowed values so that it can ensure references will be valid in more situations than it could without our help.

Removing Duplication by Extracting a Function

Generics allow us to replace specific types with a placeholder that represents multiple types to remove code duplication. Before diving into generics syntax, then, let's first look at how to remove duplication in a way that doesn't involve generic types by extracting a function that replaces specific values with a placeholder that represents multiple values. Then we'll apply the same technique to extract a generic function! By looking at how to recognize duplicated code you can extract into a function, you'll start to recognize duplicated code that can use generics.

We begin with the short program in Listing 10-1 that finds the largest number in a list.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

Listing 10-1: Finding the largest number in a list of numbers

We store a list of integers in the variable `number_list` and place the first number in the list in a variable named `largest`. We then iterate through all the numbers in the list, and if the current number is greater than the number stored in `largest`, replace the number in that variable. However, if the current number is less than or equal to the largest number seen so far, the variable doesn't change, and the code moves on to the next number in the list. After considering all the numbers in the list, `largest` should hold the largest number, which in this case is 100.

We've now been tasked with finding the largest number in two different lists of numbers. To do so, we can choose to duplicate the code in Listing 10-1 and use the same logic at two different places in the program, as shown in Listing 10-2.

Filename: src/main.rs

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = number_list[0];

    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("The largest number is {}", largest);
}
```

Listing 10-2: Code to find the largest number in *two* lists of numbers

Although this code works, duplicating code is tedious and error prone. We also have to remember to update the code in multiple places when we want to change it.

To eliminate this duplication, we'll create an abstraction by defining a function that operates on any list of integers passed in a parameter. This solution makes our code clearer and lets us express the concept of finding the largest number in a list abstractly.

In Listing 10-3, we extract the code that finds the largest number into a function named `largest`. Then we call the function to find the largest number in the two lists from Listing 10-2. We could also use the function on any other list of `i32` values we might have in the future.

Filename: src/main.rs

```
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

Listing 10-3: Abstracted code to find the largest number in two lists

The `largest` function has a parameter called `list`, which represents any concrete slice of `i32` values we might pass into the function. As a result, when we call the function, the code runs on the specific values that we pass in. Don't worry about the syntax of the `for` loop for now. We aren't referencing a reference to an `i32` here; we're pattern matching and destructuring each `&i32` that the `for` loop gets so that `item` will be an `i32` inside the loop body. We'll cover pattern matching in detail in Chapter 18.

In sum, here are the steps we took to change the code from Listing 10-2 to Listing 10-3:

1. Identify duplicate code.
2. Extract the duplicate code into the body of the function and specify the inputs and return values of that code in the function signature.
3. Update the two instances of duplicated code to call the function instead.

Next, we'll use these same steps with generics to reduce code duplication. In the same way that the function body can operate on an abstract `list` instead of specific values, generics allow code to operate on abstract types.

For example, say we had two functions: one that finds the largest item in a slice of `i32` values and one that finds the largest item in a slice of `char` values. How would we eliminate that duplication? Let's find out!

Generic Data Types

We use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types. Let’s first look at how to define functions, structs, enums, and methods using generics. Then we’ll discuss how generics affect code performance.

In Function Definitions

When defining a function that uses generics, we place the generics in the signature of the function where we would usually specify the data types of the parameters and return value. Doing so makes our code more flexible and provides more functionality to callers of our function while preventing code duplication.

Continuing with our `largest` function, Listing 10-4 shows two functions that both find the largest value in a slice. We'll then combine these into a single function that uses generics.

Filename: src/main.rs

```
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

Listing 10-4: Two functions that differ only in their names and the types in their signatures

The `largest_i32` function is the one we extracted in Listing 10-3 that finds the largest `i32` in a slice. The `largest_char` function finds the largest `char` in a slice. The function bodies have the same code, so let's eliminate the duplication by introducing a generic type parameter in a single function.

To parameterize the types in a new single function, we need to name the type parameter, just as we do for the value parameters to a function. You can use any identifier as a type parameter name. But we'll use `T` because, by convention, parameter names in Rust are short, often just a letter, and Rust's type-naming convention is CamelCase. Short for "type," `T` is the default choice of most Rust programmers.

When we use a parameter in the body of the function, we have to declare the parameter name in the signature so the compiler knows what that name means. Similarly, when we use a type parameter name in a function signature, we have to declare the type parameter name before we use it. To define the generic `largest` function, place type name declarations inside angle brackets, `<>`, between the name of the function and the parameter list, like this:

```
fn largest<T>(list: &[T]) -> T {
```

We read this definition as: the function `largest` is generic over some type `T`. This function has one parameter named `list`, which is a slice of values of type `T`. The `largest` function will return a value of the same type `T`.

Listing 10-5 shows the combined `largest` function definition using the generic data type in its signature. The listing also shows how we can call the function with either a slice of `i32` values or `char` values. Note that this code won't compile yet, but we'll fix it later in this chapter.

Filename: src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Listing 10-5: The `largest` function using generic type parameters; this doesn't yet compile

If we compile this code right now, we'll get this error:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- T
  |            |
  |            T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

The note mentions `std::cmp::PartialOrd`, which is a *trait*. We'll talk about traits in the next section. For now, know that this error states that the body of `largest` won't work for all possible types that `T` could be. Because we want to compare values of type `T` in the body, we can only use types whose values can be ordered. To enable comparisons, the standard library has the `std::cmp::PartialOrd` trait that you can implement on types (see Appendix C for more on this trait). You'll learn how to specify that a generic type has a particular trait in the "Traits as Parameters" section. Before we fix this code (in the section "Fixing the `largest` Function with Trait Bounds"), let's first explore other ways of using generic type parameters.

In Struct Definitions

We can also define structs to use a generic type parameter in one or more fields using the `<>` syntax. Listing 10-6 defines a `Point<T>` struct to hold `x` and `y` coordinate values of any type.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Listing 10-6: A `Point<T>` struct that holds `x` and `y` values of type `T`

The syntax for using generics in struct definitions is similar to that used in function definitions. First, we declare the name of the type parameter inside angle brackets just after the name of the struct. Then we use the generic type in the struct definition where we would otherwise specify concrete data types.

Note that because we've used only one generic type to define `Point<T>`, this definition says that the `Point<T>` struct is generic over some type `T`, and the fields `x` and `y` are *both* that same type, whatever that type may be. If we create an instance of a `Point<T>` that has values of different types, as in Listing 10-7, our code won't compile.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Listing 10-7: The fields `x` and `y` must be the same type because both have the same generic data type `T`.

In this example, when we assign the integer value 5 to `x`, we let the compiler know that the generic type `T` will be an integer for this instance of `Point<T>`. Then when we specify 4.0 for `y`, which we've defined to have the same type as `x`, we'll get a type mismatch error like this:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found floating-point
number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

To define a `Point` struct where `x` and `y` are both generics but could have different types, we can use multiple generic type parameters. For example, in Listing 10-8, we change the definition of `Point` to be generic over types `T` and `U` where `x` is of type `T` and `y` is of type `U`.

Filename: src/main.rs

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Listing 10-8: A `Point<T, U>` generic over two types so that `x` and `y` can be values of different types

Now all the instances of `Point` shown are allowed! You can use as many generic type parameters in a definition as you want, but using more than a few makes your code hard to read. If you're finding you need lots of generic types in your code, it could indicate that your code needs restructuring into smaller pieces.

In Enum Definitions

As we did with structs, we can define enums to hold generic data types in their variants. Let's take another look at the `Option<T>` enum that the standard library provides, which we used in Chapter 6:

```
enum Option<T> {
    Some(T),
    None,
}
```

This definition should now make more sense to you. As you can see, the `Option<T>` enum is generic over type `T` and has two variants: `Some`, which holds one value of type `T`, and a `None` variant that doesn't hold any value. By using the `Option<T>` enum, we can express the abstract concept of an optional value, and because `Option<T>` is generic, we can use this abstraction no matter what the type of the optional value is.

Enums can use multiple generic types as well. The definition of the `Result` enum that we used in Chapter 9 is one example:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `Result` enum is generic over two types, `T` and `E`, and has two variants: `Ok`, which holds a value of type `T`, and `Err`, which holds a value of type `E`. This definition makes it convenient to use the `Result` enum anywhere we have an operation that might succeed (return a value of some type `T`) or fail (return an error of some type `E`). In fact, this is what we used to open a file in Listing 9-3, where `T` was filled in with the type `std::fs::File` when the file was opened successfully and `E` was filled in with the type `std::io::Error` when there were problems opening the file.

When you recognize situations in your code with multiple struct or enum definitions that differ only in the types of the values they hold, you can avoid duplication by using generic types instead.

In Method Definitions

We can implement methods on structs and enums (as we did in Chapter 5) and use generic types in their definitions, too. Listing 10-9 shows the `Point<T>` struct we defined in Listing 10-6 with a method named `x` implemented on it.

Filename: src/main.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Listing 10-9: Implementing a method named `x` on the `Point<T>` struct that will return a reference to the `x` field of type `T`

Here, we've defined a method named `x` on `Point<T>` that returns a reference to the data in the field `x`.

Note that we have to declare `T` just after `impl` so we can use `T` to specify that we're implementing methods on the type `Point<T>`. By declaring `T` as a generic type after `impl`, Rust can identify that the type in the angle brackets in `Point` is a generic type rather than a concrete type. We could have chosen a different name for this generic parameter than the generic parameter declared in the struct definition, but using the same name is conventional. Methods written within an `impl` that declares the generic type will be defined on any instance of the type, no matter what concrete type ends up substituting for the generic type.

We can also specify constraints on generic types when defining methods on the type. We could, for example, implement methods only on `Point<f32>` instances rather than on `Point<T>` instances

with any generic type. In Listing 10-10 we use the concrete type `f32`, meaning we don't declare any types after `impl`.

Filename: src/main.rs

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

Listing 10-10: An `impl` block that only applies to a struct with a particular concrete type for the generic type parameter `T`

This code means the type `Point<f32>` will have a `distance_from_origin` method; other instances of `Point<T>` where `T` is not of type `f32` will not have this method defined. The method measures how far our point is from the point at coordinates (0.0, 0.0) and uses mathematical operations that are available only for floating point types.

Generic type parameters in a struct definition aren't always the same as those you use in that same struct's method signatures. Listing 10-11 uses the generic types `X1` and `Y1` for the `Point` struct and `X2` `Y2` for the `mixup` method signature to make the example clearer. The method creates a new `Point` instance with the `x` value from the `self` `Point` (of type `X1`) and the `y` value from the passed-in `Point` (of type `Y2`).

Filename: src/main.rs

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Listing 10-11: A method that uses generic types different from its struct's definition

In `main`, we've defined a `Point` that has an `i32` for `x` (with value `5`) and an `f64` for `y` (with value `10.4`). The `p2` variable is a `Point` struct that has a string slice for `x` (with value `"Hello"`) and a `char` for `y` (with value `c`). Calling `mixup` on `p1` with the argument `p2` gives us `p3`, which will have an `i32` for `x`, because `x` came from `p1`. The `p3` variable will have a `char` for `y`, because `y` came from `p2`. The `println!` macro call will print `p3.x = 5, p3.y = c`.

The purpose of this example is to demonstrate a situation in which some generic parameters are declared with `impl` and some are declared with the method definition. Here, the generic parameters `X1` and `Y1` are declared after `impl` because they go with the struct definition. The

generic parameters `X2` and `Y2` are declared after `fn mixup`, because they're only relevant to the method.

Performance of Code Using Generics

You might be wondering whether there is a runtime cost when using generic type parameters. The good news is that using generic types won't make your run any slower than it would with concrete types.

Rust accomplishes this by performing monomorphization of the code using generics at compile time. *Monomorphization* is the process of turning generic code into specific code by filling in the concrete types that are used when compiled. In this process, the compiler does the opposite of the steps we used to create the generic function in Listing 10-5: the compiler looks at all the places where generic code is called and generates code for the concrete types the generic code is called with.

Let's look at how this works by using the standard library's generic `Option<T>` enum:

```
let integer = Some(5);
let float = Some(5.0);
```

When Rust compiles this code, it performs monomorphization. During that process, the compiler reads the values that have been used in `Option<T>` instances and identifies two kinds of `Option<T>`: one is `i32` and the other is `f64`. As such, it expands the generic definition of `Option<T>` into `Option_i32` and `Option_f64`, thereby replacing the generic definition with the specific ones.

The monomorphized version of the code looks like the following:

Filename: src/main.rs

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

The generic `Option<T>` is replaced with the specific definitions created by the compiler. Because Rust compiles generic code into code that specifies the type in each instance, we pay no runtime cost for using generics. When the code runs, it performs just as it would if we had duplicated each definition by hand. The process of monomorphization makes Rust's generics extremely efficient at runtime.

Traits: Defining Shared Behavior

A *trait* defines functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use *trait bounds* to specify that a generic type can be any type that has certain behavior.

> Note: Traits are similar to a feature often called *interfaces* in other languages, although with some differences.

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

For example, let's say we have multiple structs that hold various kinds and amounts of text: a `NewsArticle` struct that holds a news story filed in a particular location and a `Tweet` that can have at most 280 characters along with metadata that indicates whether it was a new tweet, a retweet, or a reply to another tweet.

We want to make a media aggregator library crate named `aggregator` that can display summaries of data that might be stored in a `NewsArticle` or `Tweet` instance. To do this, we need a summary from each type, and we'll request that summary by calling a `summarize` method on an instance. Listing 10-12 shows the definition of a public `Summary` trait that expresses this behavior.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

Listing 10-12: A `Summary` trait that consists of the behavior provided by a `summarize` method

Here, we declare a trait using the `trait` keyword and then the trait's name, which is `Summary` in this case. We've also declared the trait as `pub` so that crates depending on this crate can make use of this trait too, as we'll see in a few examples. Inside the curly brackets, we declare the method signatures that describe the behaviors of the types that implement this trait, which in this case is `fn summarize(&self) -> String`.

After the method signature, instead of providing an implementation within curly brackets, we use a semicolon. Each type implementing this trait must provide its own custom behavior for the body of the method. The compiler will enforce that any type that has the `Summary` trait will have the method `summarize` defined with this signature exactly.

A trait can have multiple methods in its body: the method signatures are listed one per line and each line ends in a semicolon.

Implementing a Trait on a Type

Now that we've defined the desired signatures of the `Summary` trait's methods, we can implement it on the types in our media aggregator. Listing 10-13 shows an implementation of the `Summary` trait on the `NewsArticle` struct that uses the headline, the author, and the location to create the return value of `summarize`. For the `Tweet` struct, we define `summarize` as the username followed by the entire text of the tweet, assuming that tweet content is already limited to 280 characters.

Filename: src/lib.rs

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Listing 10-13: Implementing the `Summary` trait on the `NewsArticle` and `Tweet` types

Implementing a trait on a type is similar to implementing regular methods. The difference is that after `impl`, we put the trait name we want to implement, then use the `for` keyword, and then specify the name of the type we want to implement the trait for. Within the `impl` block, we put the method signatures that the trait definition has defined. Instead of adding a semicolon after each signature, we use curly brackets and fill in the method body with the specific behavior that we want the methods of the trait to have for the particular type.

Now that the library has implemented the `Summary` trait on `NewsArticle` and `Tweet`, users of the crate can call the trait methods on instances of `NewsArticle` and `Tweet` in the same way we call regular methods. The only difference is that the user must bring the trait into scope as well as the types. Here's an example of how a binary crate could use our `aggregator` library crate:

```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

This code prints `1 new tweet: horse_ebooks: of course, as you probably already know, people`.

Other crates that depend on the `aggregator` crate can also bring the `Summary` trait into scope to implement `Summary` on their own types. One restriction to note is that we can implement a trait on a type only if at least one of the trait or the type is local to our crate. For example, we can implement standard library traits like `Display` on a custom type like `Tweet` as part of our `aggregator` crate functionality, because the type `Tweet` is local to our `aggregator` crate. We can also implement `Summary` on `Vec<T>` in our `aggregator` crate, because the trait `Summary` is local to our `aggregator` crate.

But we can't implement external traits on external types. For example, we can't implement the `Display` trait on `Vec<T>` within our `aggregator` crate, because `Display` and `Vec<T>` are both defined in the standard library and aren't local to our `aggregator` crate. This restriction is part of a

property called *coherence*, and more specifically the *orphan rule*, so named because the parent type is not present. This rule ensures that other people's code can't break your code and vice versa. Without the rule, two crates could implement the same trait for the same type, and Rust wouldn't know which implementation to use.

Default Implementations

Sometimes it's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type. Then, as we implement the trait on a particular type, we can keep or override each method's default behavior.

In Listing 10-14 we specify a default string for the `summarize` method of the `Summary` trait instead of only defining the method signature, as we did in Listing 10-12.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Listing 10-14: Defining a `Summary` trait with a default implementation of the `summarize` method

To use a default implementation to summarize instances of `NewsArticle`, we specify an empty `impl` block with `impl Summary for NewsArticle {}`.

Even though we're no longer defining the `summarize` method on `NewsArticle` directly, we've provided a default implementation and specified that `NewsArticle` implements the `Summary` trait. As a result, we can still call the `summarize` method on an instance of `NewsArticle`, like this:

```
    let article = NewsArticle {
        headline: String::from("Penguins win the Stanley Cup Championship!"),
        location: String::from("Pittsburgh, PA, USA"),
        author: String::from("Iceburgh"),
        content: String::from(
            "The Pittsburgh Penguins once again are the best \
             hockey team in the NHL.",
        ),
    };

    println!("New article available! {}", article.summarize());
```

This code prints `New article available! (Read more...)`.

Creating a default implementation doesn't require us to change anything about the implementation of `Summary` on `Tweet` in Listing 10-13. The reason is that the syntax for overriding a default implementation is the same as the syntax for implementing a trait method that doesn't have a default implementation.

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation. In this way, a trait can provide a lot of useful functionality and only require implementors to specify a small part of it. For example, we could define the `Summary` trait to have a `summarize_author` method whose implementation is required, and then define a `summarize` method that has a default implementation that calls the `summarize_author` method:

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

To use this version of `Summary`, we only need to define `summarize_author` when we implement the trait on a type:

```
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

After we define `summarize_author`, we can call `summarize` on instances of the `Tweet` struct, and the default implementation of `summarize` will call the definition of `summarize_author` that we've provided. Because we've implemented `summarize_author`, the `Summary` trait has given us the behavior of the `summarize` method without requiring us to write any more code.

```
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
```

This code prints `1 new tweet: (Read more from @horse_ebooks...)`.

Note that it isn't possible to call the default implementation from an overriding implementation of that same method.

Traits as Parameters

Now that you know how to define and implement traits, we can explore how to use traits to define functions that accept many different types. We'll use the `Summary` trait we implemented on the `NewsArticle` and `Tweet` types in Listing 10-13 to define a `notify` function that calls the `summarize` method on its `item` parameter, which is of some type that implements the `Summary` trait. To do this, we use the `impl Trait` syntax, like this:

```
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Instead of a concrete type for the `item` parameter, we specify the `impl` keyword and the trait name. This parameter accepts any type that implements the specified trait. In the body of `notify`, we can call any methods on `item` that come from the `Summary` trait, such as `summarize`. We can call `notify` and pass in any instance of `NewsArticle` or `Tweet`. Code that calls the function with any other type, such as a `String` or an `i32`, won't compile because those types don't implement `Summary`.

Trait Bound Syntax

The `impl Trait` syntax works for straightforward cases but is actually syntax sugar for a longer form known as a *trait bound*; it looks like this:

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

This longer form is equivalent to the example in the previous section but is more verbose. We place trait bounds with the declaration of the generic type parameter after a colon and inside angle brackets.

The `impl Trait` syntax is convenient and makes for more concise code in simple cases, while the fuller trait bound syntax can express more complexity in other cases. For example, we can have two parameters that implement `Summary`. Doing so with the `impl Trait` syntax looks like this:

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

Using `impl Trait` is appropriate if we want this function to allow `item1` and `item2` to have different types (as long as both types implement `Summary`). If we want to force both parameters to have the same type, however, we must use a trait bound, like this:

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

The generic type `T` specified as the type of the `item1` and `item2` parameters constrains the function such that the concrete type of the value passed as an argument for `item1` and `item2` must be the same.

Specifying Multiple Trait Bounds with the `+` Syntax

We can also specify more than one trait bound. Say we wanted `notify` to use display formatting as well as `summarize` on `item`: we specify in the `notify` definition that `item` must implement both `Display` and `Summary`. We can do so using the `+` syntax:

```
pub fn notify(item: &(impl Summary + Display)) {
```

The `+` syntax is also valid with trait bounds on generic types:

```
pub fn notify<T: Summary + Display>(item: &T) {
```

With the two trait bounds specified, the body of `notify` can call `summarize` and use `{}` to format `item`.

Clearer Trait Bounds with `where` Clauses

Using too many trait bounds has its downsides. Each generic has its own trait bounds, so functions with multiple generic type parameters can contain lots of trait bound information between the function's name and its parameter list, making the function signature hard to read. For this reason, Rust has alternate syntax for specifying trait bounds inside a `where` clause after the function signature. So instead of writing this:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

we can use a `where` clause, like this:

```
fn some_function<T, U>(t: &T, u: &U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

This function's signature is less cluttered: the function name, parameter list, and return type are close together, similar to a function without lots of trait bounds.

Returning Types that Implement Traits

We can also use the `impl Trait` syntax in the return position to return a value of some type that implements a trait, as shown here:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

By using `impl Summary` for the return type, we specify that the `returns_summarizable` function returns some type that implements the `Summary` trait without naming the concrete type. In this case, `returns_summarizable` returns a `Tweet`, but the code calling this function doesn't need to know that.

The ability to specify a return type only by the trait it implements is especially useful in the context of closures and iterators, which we cover in Chapter 13. Closures and iterators create types that only the compiler knows or types that are very long to specify. The `impl Trait` syntax lets you concisely specify that a function returns some type that implements the `Iterator` trait without needing to write out a very long type.

However, you can only use `impl Trait` if you're returning a single type. For example, this code that returns either a `NewsArticle` or a `Tweet` with the return type specified as `impl Summary` wouldn't work:

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Returning either a `NewsArticle` or a `Tweet` isn't allowed due to restrictions around how the `impl Trait` syntax is implemented in the compiler. We'll cover how to write a function with this behavior in the "Using Trait Objects That Allow for Values of Different Types" section of Chapter 17.

Fixing the `largest` Function with Trait Bounds

Now that you know how to specify the behavior you want using the generic type parameter's bounds, let's return to Listing 10-5 to fix the definition of the `largest` function that uses a generic type parameter! Last time we tried to run that code, we received this error:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- T
  |            |
  |            T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

In the body of `largest` we wanted to compare two values of type `T` using the greater than (`>`) operator. Because that operator is defined as a default method on the standard library trait `std::cmp::PartialOrd`, we need to specify `PartialOrd` in the trait bounds for `T` so the `largest` function can work on slices of any type that we can compare. We don't need to bring `PartialOrd` into scope because it's in the prelude. Change the signature of `largest` to look like this:

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

This time when we compile the code, we get a different set of errors:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out of here
  |                       move occurs because `list[_]` has type `T`, which does not
implement the `Copy` trait
  |                       help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of a shared reference
 --> src/main.rs:4:18
  |
4 |     for &item in list {
  |         -----    ^^^^
  |         ||
  |         |data moved here
  |         |move occurs because `item` has type `T`, which does not implement the
`Copy` trait
  |         help: consider removing the `&`: `item`

Some errors have detailed explanations: E0507, E0508.
For more information about an error, try `rustc --explain E0507`.
error: could not compile `chapter10` due to 2 previous errors
```

The key line in this error is `cannot move out of type [T], a non-copy slice`. With our non-generic versions of the `largest` function, we were only trying to find the largest `i32` or `char`. As discussed in the "Stack-Only Data: Copy" section in Chapter 4, types like `i32` and `char` that have a known size can be stored on the stack, so they implement the `Copy` trait. But when we made the `largest` function generic, it became possible for the `list` parameter to have types in it that don't implement the `Copy` trait. Consequently, we wouldn't be able to move the value out of `list[0]` and into the `largest` variable, resulting in this error.

To call this code with only those types that implement the `Copy` trait, we can add `Copy` to the trait bounds of `T`! Listing 10-15 shows the complete code of a generic `largest` function that will compile as long as the types of the values in the slice that we pass into the function implement the `PartialOrd` *and* `Copy` traits, like `i32` and `char` do.

Filename: src/main.rs

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Listing 10-15: A working definition of the `largest` function that works on any generic type that implements the `PartialOrd` and `Copy` traits

If we don't want to restrict the `largest` function to the types that implement the `Copy` trait, we could specify that `T` has the trait bound `Clone` instead of `Copy`. Then we could clone each value in

the slice when we want the `largest` function to have ownership. Using the `clone` function means we're potentially making more heap allocations in the case of types that own heap data like `String`, and heap allocations can be slow if we're working with large amounts of data.

We could also implement `largest` by having the function return a reference to a `T` value in the slice. If we change the return type to `&T` instead of `T`, thereby changing the body of the function to return a reference, we wouldn't need the `Clone` or `Copy` trait bounds and we could avoid heap allocations. Try implementing these alternate solutions on your own! If you get stuck with errors having to do with lifetimes, keep reading: the "Validating References with Lifetimes" section coming up will explain, but lifetimes aren't required to solve these challenges.

Using Trait Bounds to Conditionally Implement Methods

By using a trait bound with an `impl` block that uses generic type parameters, we can implement methods conditionally for types that implement the specified traits. For example, the type `Pair<T>` in Listing 10-16 always implements the `new` function to return a new instance of `Pair<T>` (recall from the "Defining Methods" section of Chapter 5 that `Self` is a type alias for the type of the `impl` block, which in this case is `Pair<T>`). But in the next `impl` block, `Pair<T>` only implements the `cmp_display` method if its inner type `T` implements the `PartialOrd` trait that enables comparison *and* the `Display` trait that enables printing.

Filename: src/lib.rs

```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Listing 10-16: Conditionally implement methods on a generic type depending on trait bounds

We can also conditionally implement a trait for any type that implements another trait. Implementations of a trait on any type that satisfies the trait bounds are called *blanket implementations* and are extensively used in the Rust standard library. For example, the standard library implements the `ToString` trait on any type that implements the `Display` trait. The `impl` block in the standard library looks similar to this code:

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Because the standard library has this blanket implementation, we can call the `to_string` method defined by the `ToString` trait on any type that implements the `Display` trait. For example, we can turn integers into their corresponding `String` values like this because integers implement `Display`:

```
let s = 3.to_string();
```

Blanket implementations appear in the documentation for the trait in the “Implementors” section.

Traits and trait bounds let us write code that uses generic type parameters to reduce duplication but also specify to the compiler that we want the generic type to have particular behavior. The compiler can then use the trait bound information to check that all the concrete types used with our code provide the correct behavior. In dynamically typed languages, we would get an error at runtime if we called a method on a type which didn’t define the method. But Rust moves these errors to compile time so we’re forced to fix the problems before our code is even able to run. Additionally, we don’t have to write code that checks for behavior at runtime because we’ve already checked at compile time. Doing so improves performance without having to give up the flexibility of generics.

Validating References with Lifetimes

Lifetimes are another kind of generic that we've already been using. Rather than ensuring that a type has the behavior we want, lifetimes ensure that references are valid as long as we need them to be.

One detail we didn't discuss in the "References and Borrowing" section in Chapter 4 is that every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. Most of the time, lifetimes are implicit and inferred, just like most of the time, types are inferred. We only must annotate types when multiple types are possible. In a similar way, we must annotate lifetimes when the lifetimes of references could be related in a few different ways. Rust requires us to annotate the relationships using generic lifetime parameters to ensure the actual references used at runtime will definitely be valid.

Annotating lifetimes is not even a concept most other programming languages have, so this is going to feel unfamiliar. Although we won't cover lifetimes in their entirety in this chapter, we'll discuss common ways you might encounter lifetime syntax so you can get comfortable with the concept.

Preventing Dangling References with Lifetimes

The main aim of lifetimes is to prevent *dangling references*, which cause a program to reference data other than the data it's intended to reference. Consider the program in Listing 10-17, which has an outer scope and an inner scope.

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

Listing 10-17: An attempt to use a reference whose value has gone out of scope

---

Note: The examples in Listings 10-17, 10-18, and 10-24 declare variables without giving them an initial value, so the variable name exists in the outer scope. At first glance, this might appear to be in conflict with Rust's having no null values. However, if we try to use a variable before giving it a value, we'll get a compile-time error, which shows that Rust indeed does not allow null values.

The outer scope declares a variable named `r` with no initial value, and the inner scope declares a variable named `x` with the initial value of 5. Inside the inner scope, we attempt to set the value of `r` as a reference to `x`. Then the inner scope ends, and we attempt to print the value in `r`. This code won't compile because the value `r` is referring to has gone out of scope before we try to use it. Here is the error message:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
  --> src/main.rs:7:17
   |
7  |             r = &x;
   |                 ^^ borrowed value does not live long enough
8  |         }
   |         - `x` dropped here while still borrowed
9  |
10 |         println!("r: {}", r);
   |                           - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

The variable `x` doesn't "live long enough." The reason is that `x` will be out of scope when the inner scope ends on line 7. But `r` is still valid for the outer scope; because its scope is larger, we say that it "lives longer." If Rust allowed this code to work, `r` would be referencing memory that was deallocated when `x` went out of scope, and anything we tried to do with `r` wouldn't work correctly. So how does Rust determine that this code is invalid? It uses a borrow checker.

The Borrow Checker

The Rust compiler has a *borrow checker* that compares scopes to determine whether all borrows are valid. Listing 10-18 shows the same code as Listing 10-17 but with annotations showing the lifetimes of the variables.

```
{
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

Listing 10-18: Annotations of the lifetimes of `r` and `x`, named `'a` and `'b`, respectively

Here, we've annotated the lifetime of `r` with `'a` and the lifetime of `x` with `'b`. As you can see, the inner `'b` block is much smaller than the outer `'a` lifetime block. At compile time, Rust compares the size of the two lifetimes and sees that `r` has a lifetime of `'a` but that it refers to memory with a lifetime of `'b`. The program is rejected because `'b` is shorter than `'a`: the subject of the reference doesn't live as long as the reference.

Listing 10-19 fixes the code so it doesn't have a dangling reference and compiles without any errors.

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

Listing 10-19: A valid reference because the data has a longer lifetime than the reference

Here, `x` has the lifetime `'b`, which in this case is larger than `'a`. This means `r` can reference `x` because Rust knows that the reference in `r` will always be valid while `x` is valid.

Now that you know where the lifetimes of references are and how Rust analyzes lifetimes to ensure references will always be valid, let's explore generic lifetimes of parameters and return values in the context of functions.

Generic Lifetimes in Functions

We'll write a function that returns the longer of two string slices. This function will take two string slices and return a single string slice. After we've implemented the `longest` function, the code in Listing 10-20 should print `The longest string is abcd`.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

Listing 10-20: A `main` function that calls the `longest` function to find the longer of two string slices

Note that we want the function to take string slices, which are references, rather than strings, because we don't want the `longest` function to take ownership of its parameters. Refer to the "String Slices as Parameters" section in Chapter 4 for more discussion about why the parameters we use in Listing 10-20 are the ones we want.

If we try to implement the `longest` function as shown in Listing 10-21, it won't compile.

Filename: src/main.rs

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-21: An implementation of the `longest` function that returns the longer of two string slices but does not yet compile

Instead, we get the following error that talks about lifetimes:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does
not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

The help text reveals that the return type needs a generic lifetime parameter on it because Rust can't tell whether the reference being returned refers to `x` or `y`. Actually, we don't know either, because the `if` block in the body of this function returns a reference to `x` and the `else` block returns a reference to `y`!

When we're defining this function, we don't know the concrete values that will be passed into this function, so we don't know whether the `if` case or the `else` case will execute. We also don't know the concrete lifetimes of the references that will be passed in, so we can't look at the scopes as we did in Listings 10-18 and 10-19 to determine whether the reference we return will always be valid. The borrow checker can't determine this either, because it doesn't know how the lifetimes of `x` and `y` relate to the lifetime of the return value. To fix this error, we'll add generic lifetime parameters that define the relationship between the references so the borrow checker can perform its analysis.

Lifetime Annotation Syntax

Lifetime annotations don't change how long any of the references live. Rather, they describe the relationships of the lifetimes of multiple references to each other without affecting the lifetimes. Just as functions can accept any type when the signature specifies a generic type parameter, functions can accept references with any lifetime by specifying a generic lifetime parameter.

Lifetime annotations have a slightly unusual syntax: the names of lifetime parameters must start with an apostrophe (`'`) and are usually all lowercase and very short, like generic types. Most people use the name `'a` for the first lifetime annotation. We place lifetime parameter annotations after the `&` of a reference, using a space to separate the annotation from the reference's type.

Here are some examples: a reference to an `i32` without a lifetime parameter, a reference to an `i32` that has a lifetime parameter named `'a`, and a mutable reference to an `i32` that also has the lifetime `'a`.

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

One lifetime annotation by itself doesn't have much meaning, because the annotations are meant to tell Rust how generic lifetime parameters of multiple references relate to each other. For example,

let's say we have a function with the parameter `first` that is a reference to an `i32` with lifetime `'a`. The function also has another parameter named `second` that is another reference to an `i32` that also has the lifetime `'a`. The lifetime annotations indicate that the references `first` and `second` must both live as long as that generic lifetime.

Lifetime Annotations in Function Signatures

Now let's examine lifetime annotations in the context of the `longest` function. As with generic type parameters, we need to declare generic lifetime parameters inside angle brackets between the function name and the parameter list. We want the signature to express the following constraint: the returned reference will be valid as long as both the parameters are valid. This is the relationship between lifetimes of the parameters and the return value. We'll name the lifetime `'a` and then add it to each reference, as shown in Listing 10-22.

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 10-22: The `longest` function definition specifying that all the references in the signature must have the same lifetime `'a`

This code should compile and produce the result we want when we use it with the `main` function in Listing 10-20.

The function signature now tells Rust that for some lifetime `'a`, the function takes two parameters, both of which are string slices that live at least as long as lifetime `'a`. The function signature also tells Rust that the string slice returned from the function will live at least as long as lifetime `'a`. In practice, it means that the lifetime of the reference returned by the `longest` function is the same as the smaller of the lifetimes of the references passed in. These relationships are what we want Rust to use when analyzing this code.

Remember, when we specify the lifetime parameters in this function signature, we're not changing the lifetimes of any values passed in or returned. Rather, we're specifying that the borrow checker should reject any values that don't adhere to these constraints. Note that the `longest` function doesn't need to know exactly how long `x` and `y` will live, only that some scope can be substituted for `'a` that will satisfy this signature.

When annotating lifetimes in functions, the annotations go in the function signature, not in the function body. The lifetime annotations become part of the contract of the function, much like the types in the signature. Having function signatures contain the lifetime contract means the analysis the Rust compiler does can be simpler. If there's a problem with the way a function is annotated or the way it is called, the compiler errors can point to the part of our code and the constraints more precisely. If, instead, the Rust compiler made more inferences about what we intended the relationships of the lifetimes to be, the compiler might only be able to point to a use of our code many steps away from the cause of the problem.

When we pass concrete references to `longest`, the concrete lifetime that is substituted for `'a` is the part of the scope of `x` that overlaps with the scope of `y`. In other words, the generic lifetime `'a` will get the concrete lifetime that is equal to the smaller of the lifetimes of `x` and `y`. Because we've annotated the returned reference with the same lifetime parameter `'a`, the returned reference will also be valid for the length of the smaller of the lifetimes of `x` and `y`.

Let's look at how the lifetime annotations restrict the `longest` function by passing in references that have different concrete lifetimes. Listing 10-23 is a straightforward example.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

Listing 10-23: Using the `longest` function with references to `String` values that have different concrete lifetimes

In this example, `string1` is valid until the end of the outer scope, `string2` is valid until the end of the inner scope, and `result` references something that is valid until the end of the inner scope. Run this code, and you'll see that the borrow checker approves; it will compile and print `The longest string is long string is long`.

Next, let's try an example that shows that the lifetime of the reference in `result` must be the smaller lifetime of the two arguments. We'll move the declaration of the `result` variable outside the inner scope but leave the assignment of the value to the `result` variable inside the scope with `string2`. Then we'll move the `println!` that uses `result` to outside the inner scope, after the inner scope has ended. The code in Listing 10-24 will not compile.

Filename: src/main.rs

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

Listing 10-24: Attempting to use `result` after `string2` has gone out of scope

When we try to compile this code, we get this error:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed value does not
live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
  |                                          ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

The error shows that for `result` to be valid for the `println!` statement, `string2` would need to be valid until the end of the outer scope. Rust knows this because we annotated the lifetimes of the function parameters and return values using the same lifetime parameter `'a`.

As humans, we can look at this code and see that `string1` is longer than `string2` and therefore `result` will contain a reference to `string1`. Because `string1` has not gone out of scope yet, a

reference to `string1` will still be valid for the `println!` statement. However, the compiler can't see that the reference is valid in this case. We've told Rust that the lifetime of the reference returned by the `longest` function is the same as the smaller of the lifetimes of the references passed in. Therefore, the borrow checker disallows the code in Listing 10-24 as possibly having an invalid reference.

Try designing more experiments that vary the values and lifetimes of the references passed in to the `longest` function and how the returned reference is used. Make hypotheses about whether or not your experiments will pass the borrow checker before you compile; then check to see if you're right!

Thinking in Terms of Lifetimes

The way in which you need to specify lifetime parameters depends on what your function is doing. For example, if we changed the implementation of the `longest` function to always return the first parameter rather than the longest string slice, we wouldn't need to specify a lifetime on the `y` parameter. The following code will compile:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

We've specified a lifetime parameter `'a` for the parameter `x` and the return type, but not for the parameter `y`, because the lifetime of `y` does not have any relationship with the lifetime of `x` or the return value.

When returning a reference from a function, the lifetime parameter for the return type needs to match the lifetime parameter for one of the parameters. If the reference returned does *not* refer to one of the parameters, it must refer to a value created within this function. However, this would be a

dangling reference because the value will go out of scope at the end of the function. Consider this attempted implementation of the `longest` function that won't compile:

Filename: src/main.rs

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

Here, even though we've specified a lifetime parameter `'a` for the return type, this implementation will fail to compile because the return value lifetime is not related to the lifetime of the parameters at all. Here is the error message we get:

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

The problem is that `result` goes out of scope and gets cleaned up at the end of the `longest` function. We're also trying to return a reference to `result` from the function. There is no way we can specify lifetime parameters that would change the dangling reference, and Rust won't let us create a dangling reference. In this case, the best fix would be to return an owned data type rather than a reference so the calling function is then responsible for cleaning up the value.

Ultimately, lifetime syntax is about connecting the lifetimes of various parameters and return values of functions. Once they're connected, Rust has enough information to allow memory-safe operations and disallow operations that would create dangling pointers or otherwise violate memory safety.

Lifetime Annotations in Struct Definitions

So far, the structs we've define all hold owned types. We can define structs to hold references, but in that case we would need to add a lifetime annotation on every reference in the struct's definition. Listing 10-25 has a struct named `ImportantExcerpt` that holds a string slice.

Filename: src/main.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Listing 10-25: A struct that holds a reference, so its definition needs a lifetime annotation

This struct has one field, `part`, that holds a string slice, which is a reference. As with generic data types, we declare the name of the generic lifetime parameter inside angle brackets after the name of the struct so we can use the lifetime parameter in the body of the struct definition. This annotation means an instance of `ImportantExcerpt` can't outlive the reference it holds in its `part` field.

The `main` function here creates an instance of the `ImportantExcerpt` struct that holds a reference to the first sentence of the `String` owned by the variable `novel`. The data in `novel` exists before the `ImportantExcerpt` instance is created. In addition, `novel` doesn't go out of scope until after the `ImportantExcerpt` goes out of scope, so the reference in the `ImportantExcerpt` instance is valid.

Lifetime Elision

You've learned that every reference has a lifetime and that you need to specify lifetime parameters for functions or structs that use references. However, in Chapter 4 we had a function in Listing 4-9, shown again in Listing 10-26, that compiled without lifetime annotations.

Filename: src/lib.rs

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Listing 10-26: A function we defined in Listing 4-9 that compiled without lifetime annotations, even though the parameter and return type are references

The reason this function compiles without lifetime annotations is historical: in early versions (pre-1.0) of Rust, this code wouldn't have compiled because every reference needed an explicit lifetime. At that time, the function signature would have been written like this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

After writing a lot of Rust code, the Rust team found that Rust programmers were entering the same lifetime annotations over and over in particular situations. These situations were predictable and followed a few deterministic patterns. The developers programmed these patterns into the compiler's code so the borrow checker could infer the lifetimes in these situations and wouldn't need explicit annotations.

This piece of Rust history is relevant because it's possible that more deterministic patterns will emerge and be added to the compiler. In the future, even fewer lifetime annotations might be required.

The patterns programmed into Rust's analysis of references are called the *lifetime elision rules*. These aren't rules for programmers to follow; they're a set of particular cases that the compiler will consider, and if your code fits these cases, you don't need to write the lifetimes explicitly.

The elision rules don't provide full inference. If Rust deterministically applies the rules but there is still ambiguity as to what lifetimes the references have, the compiler won't guess what the lifetime of the remaining references should be. Instead of guessing, the compiler will give you an error that you can resolve by adding the lifetime annotations.

Lifetimes on function or method parameters are called *input lifetimes*, and lifetimes on return values are called *output lifetimes*.

The compiler uses three rules to figure out the lifetimes of the references when there aren't explicit annotations. The first rule applies to input lifetimes, and the second and third rules apply to output lifetimes. If the compiler gets to the end of the three rules and there are still references for which it can't figure out lifetimes, the compiler will stop with an error. These rules apply to `fn` definitions as well as `impl` blocks.

The first rule is that the compiler assigns a lifetime parameter to each parameter that's a reference. In other words, a function with one parameter gets one lifetime parameter: `fn foo<'a>(x: &'a i32)`; a function with two parameters gets two separate lifetime parameters: `fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`; and so on.

The second rule is that, if there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters: `fn foo<'a>(x: &'a i32) -> &'a i32`.

The third rule is that, if there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method, the lifetime of `self` is assigned to all output lifetime

parameters. This third rule makes methods much nicer to read and write because fewer symbols are necessary.

Let's pretend we're the compiler. We'll apply these rules to figure out the lifetimes of the references in the signature of the `first_word` function in Listing 10-26. The signature starts without any lifetimes associated with the references:

```
fn first_word(s: &str) -> &str {
```

Then the compiler applies the first rule, which specifies that each parameter gets its own lifetime. We'll call it `'a` as usual, so now the signature is this:

```
fn first_word<'a>(s: &'a str) -> &str {
```

The second rule applies because there is exactly one input lifetime. The second rule specifies that the lifetime of the one input parameter gets assigned to the output lifetime, so the signature is now this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Now all the references in this function signature have lifetimes, and the compiler can continue its analysis without needing the programmer to annotate the lifetimes in this function signature.

Let's look at another example, this time using the `longest` function that had no lifetime parameters when we started working with it in Listing 10-21:

```
fn longest(x: &str, y: &str) -> &str {
```

Let's apply the first rule: each parameter gets its own lifetime. This time we have two parameters instead of one, so we have two lifetimes:

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

You can see that the second rule doesn't apply because there is more than one input lifetime. The third rule doesn't apply either, because `longest` is a function rather than a method, so none of the parameters are `self`. After working through all three rules, we still haven't figured out what the return type's lifetime is. This is why we got an error trying to compile the code in Listing 10-21: the compiler worked through the lifetime elision rules but still couldn't figure out all the lifetimes of the references in the signature.

Because the third rule really only applies in method signatures, we'll look at lifetimes in that context next to see why the third rule means we don't have to annotate lifetimes in method signatures very often.

Lifetime Annotations in Method Definitions

When we implement methods on a struct with lifetimes, we use the same syntax as that of generic type parameters shown in Listing 10-11. Where we declare and use the lifetime parameters depends on whether they're related to the struct fields or the method parameters and return values.

Lifetime names for struct fields always need to be declared after the `impl` keyword and then used after the struct's name, because those lifetimes are part of the struct's type.

In method signatures inside the `impl` block, references might be tied to the lifetime of references in the struct's fields, or they might be independent. In addition, the lifetime elision rules often make it so that lifetime annotations aren't necessary in method signatures. Let's look at some examples using the struct named `ImportantExcerpt` that we defined in Listing 10-25.

First, we'll use a method named `level` whose only parameter is a reference to `self` and whose return value is an `i32`, which is not a reference to anything:

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

The lifetime parameter declaration after `impl` and its use after the type name are required, but we're not required to annotate the lifetime of the reference to `self` because of the first elision rule.

Here is an example where the third lifetime elision rule applies:

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

There are two input lifetimes, so Rust applies the first lifetime elision rule and gives both `&self` and `announcement` their own lifetimes. Then, because one of the parameters is `&self`, the return type gets the lifetime of `&self`, and all lifetimes have been accounted for.

The Static Lifetime

One special lifetime we need to discuss is `'static`, which denotes that the affected reference *can* live for the entire duration of the program. All string literals have the `'static` lifetime, which we can annotate as follows:

```
let s: &'static str = "I have a static lifetime.";
```

The text of this string is stored directly in the program's binary, which is always available. Therefore, the lifetime of all string literals is `'static`.

You might see suggestions to use the `'static` lifetime in error messages. But before specifying `'static` as the lifetime for a reference, think about whether the reference you have actually lives the entire lifetime of your program or not, and whether you want it to. Most of the time, an error message suggesting the `'static` lifetime results from attempting to create a dangling reference or a mismatch of the available lifetimes. In such cases, the solution is fixing those problems, not specifying the `'static` lifetime.

Generic Type Parameters, Trait Bounds, and Lifetimes Together

Let's briefly look at the syntax of specifying generic type parameters, trait bounds, and lifetimes all in one function!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This is the `longest` function from Listing 10-22 that returns the longer of two string slices. But now it has an extra parameter named `ann` of the generic type `T`, which can be filled in by any type that implements the `Display` trait as specified by the `where` clause. This extra parameter will be printed using `{}`, which is why the `Display` trait bound is necessary. Because lifetimes are a type of generic, the declarations of the lifetime parameter `'a` and the generic type parameter `T` go in the same list inside the angle brackets after the function name.

Summary

We covered a lot in this chapter! Now that you know about generic type parameters, traits and trait bounds, and generic lifetime parameters, you're ready to write code without repetition that works in many different situations. Generic type parameters let you apply the code to different types. Traits and trait bounds ensure that even though the types are generic, they'll have the behavior the code needs. You learned how to use lifetime annotations to ensure that this flexible code won't have any

dangling references. And all of this analysis happens at compile time, which doesn't affect runtime performance!

Believe it or not, there is much more to learn on the topics we discussed in this chapter: Chapter 17 discusses trait objects, which are another way to use traits. There are also more complex scenarios involving lifetime annotations that you will only need in very advanced scenarios; for those, you should read the Rust Reference. But next, you'll learn how to write tests in Rust so you can make sure your code is working the way it should.

Writing Automated Tests

In his 1972 essay "The Humble Programmer," Edsger W. Dijkstra said that "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." That doesn't mean we shouldn't try to test as much as we can!

Correctness in our programs is the extent to which our code does what we intend it to do. Rust is designed with a high degree of concern about the correctness of programs, but correctness is complex and not easy to prove. Rust's type system shoulders a huge part of this burden, but the type system cannot catch everything. As such, Rust includes support for writing automated software tests.

Say we write a function `add_two` that adds 2 to whatever number is passed to it. This function's signature accepts an integer as a parameter and returns an integer as a result. When we implement and compile that function, Rust does all the type checking and borrow checking that you've learned so far to ensure that, for instance, we aren't passing a `String` value or an invalid reference to this function. But Rust *can't* check that this function will do precisely what we intend, which is return the parameter plus 2 rather than, say, the parameter plus 10 or the parameter minus 50! That's where tests come in.

We can write tests that assert, for example, that when we pass `3` to the `add_two` function, the returned value is `5`. We can run these tests whenever we make changes to our code to make sure any existing correct behavior has not changed.

Testing is a complex skill: although we can't cover every detail about how to write good tests in one chapter, we'll discuss the mechanics of Rust's testing facilities. We'll talk about the annotations and macros available to you when writing your tests, the default behavior and options provided for running your tests, and how to organize tests into unit tests and integration tests.

How to Write Tests

Tests are Rust functions that verify that the non-test code is functioning in the expected manner. The bodies of test functions typically perform these three actions:

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you expect.

Let's look at the features Rust provides specifically for writing tests that take these actions, which include the `test` attribute, a few macros, and the `should_panic` attribute.

The Anatomy of a Test Function

At its simplest, a test in Rust is a function that's annotated with the `test` attribute. Attributes are metadata about pieces of Rust code; one example is the `derive` attribute we used with structs in Chapter 5. To change a function into a test function, add `#[test]` on the line before `fn`. When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

Whenever we make a new library project with Cargo, a test module with a test function in it is automatically generated for us. This module gives you a template for writing your tests so you don't have to look up the exact structure and syntax every time you start a new project. You can add as many additional test functions and as many test modules as you want!

We'll explore some aspects of how tests work by experimenting with the template test before we actually test any code. Then we'll write some real-world tests that call some code that we've written and assert that its behavior is correct.

Let's create a new library project called `adder` that will add two numbers:

```
$ cargo new adder --lib
     Created library `adder` project
$ cd adder
```

The contents of the *src/lib.rs* file in your `adder` library should look like Listing 11-1.

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

Listing 11-1: The test module and function generated automatically by `cargo new`

For now, let's ignore the top two lines and focus on the function. Note the `#[test]` annotation: this attribute indicates this is a test function, so the test runner knows to treat this function as a test. We might also have non-test functions in the `tests` module to help set up common scenarios or perform common operations, so we always need to indicate which functions are tests.

The example function body uses the `assert_eq!` macro to assert that `result`, which contains the result of adding 2 and 2, equals 4. This assertion serves as an example of the format for a typical test. Let's run it to see that this test passes.

The `cargo test` command runs all tests in our project, as shown in Listing 11-2.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.57s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Listing 11-2: The output from running the automatically generated test

Cargo compiled and ran the test. We see the line `running 1 test`. The next line shows the name of the generated test function, called `it_works`, and that the result of running that test is `ok`. The overall summary `test result: ok.` means that all the tests passed, and the portion that reads `1 passed; 0 failed` totals the number of tests that passed or failed.

It's possible to mark a test as ignored so it doesn't run in a particular instance; we'll cover that in the "Ignoring Some Tests Unless Specifically Requested" section later in this chapter. Because we haven't done that here, the summary shows `0 ignored`. We can also pass an argument to the `cargo test` command to run only tests whose name matches a string; this is called filtering and we'll cover that in the "Running a Subset of Tests by Name" section. We also haven't filtered the tests being run, so the end of the summary shows `0 filtered out`.

The `0 measured` statistic is for benchmark tests that measure performance. Benchmark tests are, as of this writing, only available in nightly Rust. See the documentation about benchmark tests to learn

more.

The next part of the test output starting at `Doc-tests adder` is for the results of any documentation tests. We don't have any documentation tests yet, but Rust can compile any code examples that appear in our API documentation. This feature helps keep your docs and your code in sync! We'll discuss how to write documentation tests in the "Documentation Comments as Tests" section of Chapter 14. For now, we'll ignore the `Doc-tests` output.

Let's start to customize the test to our own needs. First change the name of the `it_works` function to a different name, such as `exploration`, like so:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

Then run `cargo test` again. The output now shows `exploration` instead of `it_works`:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.59s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Now we'll add another test, but this time we'll make a test that fails! Tests fail when something in the test function panics. Each test is run in a new thread, and when the main thread sees that a test thread has died, the test is marked as failed. In Chapter 9, we talked about how the simplest way to panic is to call the `panic!` macro. Enter the new test as a function named `another`, so your *src/lib.rs* file looks like Listing 11-3.

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }

    #[test]
    fn another() {
        panic!("Make this test fail");
    }
}
```

Listing 11-3: Adding a second test that will fail because we call the `panic!` macro

Run the tests again using `cargo test`. The output should look like Listing 11-4, which shows that our `exploration` test passed and `another` failed.

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.72s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::another ... FAILED
test tests::exploration ... ok

failures:

---- tests::another stdout ----
thread 'main' panicked at 'Make this test fail', src/lib.rs:10:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Listing 11-4: Test results when one test passes and one test fails

Instead of `ok`, the line `test tests::another` shows `FAILED`. Two new sections appear between the individual results and the summary: the first displays the detailed reason for each test failure. In this case, we get the details that `another` failed because it `panicked at 'Make this test fail'` on line 10 in the *src/lib.rs* file. The next section lists just the names of all the failing tests, which is useful when there are lots of tests and lots of detailed failing test output. We can use the name of a failing test to run just that test to more easily debug it; we'll talk more about ways to run tests in the "Controlling How Tests Are Run" section.

The summary line displays at the end: overall, our test result is `FAILED`. We had one test pass and one test fail.

Now that you've seen what the test results look like in different scenarios, let's look at some macros other than `panic!` that are useful in tests.

Checking Results with the `assert!` Macro

The `assert!` macro, provided by the standard library, is useful when you want to ensure that some condition in a test evaluates to `true`. We give the `assert!` macro an argument that evaluates to a Boolean. If the value is `true`, nothing happens and the test passes. If the value is `false`, the `assert!` macro calls `panic!` to cause the test to fail. Using the `assert!` macro helps us check that our code is functioning in the way we intend.

In Chapter 5, Listing 5-15, we used a `Rectangle` struct and a `can_hold` method, which are repeated here in Listing 11-5. Let's put this code in the *src/lib.rs* file, then write some tests for it using the `assert!` macro.

Filename: src/lib.rs

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

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Listing 11-5: Using the `Rectangle` struct and its `can_hold` method from Chapter 5

The `can_hold` method returns a Boolean, which means it's a perfect use case for the `assert!` macro. In Listing 11-6, we write a test that exercises the `can_hold` method by creating a `Rectangle` instance that has a width of 8 and a height of 7 and asserting that it can hold another `Rectangle` instance that has a width of 5 and a height of 1.

Filename: src/lib.rs

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

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(larger.can_hold(&smaller));
    }
}
```

Listing 11-6: A test for `can_hold` that checks whether a larger rectangle can indeed hold a smaller rectangle

Note that we've added a new line inside the `tests` module: `use super::*;`. The `tests` module is a regular module that follows the usual visibility rules we covered in Chapter 7 in the "Paths for Referring to an Item in the Module Tree" section. Because the `tests` module is an inner module, we need to bring the code under test in the outer module into the scope of the inner module. We use a glob here so anything we define in the outer module is available to this `tests` module.

We've named our test `larger_can_hold_smaller`, and we've created the two `Rectangle` instances that we need. Then we called the `assert!` macro and passed it the result of calling `larger.can_hold(&smaller)`. This expression is supposed to return `true`, so our test should pass. Let's find out!

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

It does pass! Let's add another test, this time asserting that a smaller rectangle cannot hold a larger rectangle:

Filename: src/lib.rs

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

    #[test]
    fn larger_can_hold_smaller() {
        // --snip--
    }

    #[test]
    fn smaller_cannot_hold_larger() {
        let larger = Rectangle {
            width: 8,
            height: 7,
        };
        let smaller = Rectangle {
            width: 5,
            height: 1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

Because the correct result of the `can_hold` function in this case is `false`, we need to negate that result before we pass it to the `assert!` macro. As a result, our test will pass if `can_hold` returns `false`:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... ok
test tests::smaller_cannot_hold_larger ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests rectangle

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Two tests that pass! Now let's see what happens to our test results when we introduce a bug in our code. We'll change the implementation of the `can_hold` method by replacing the greater-than sign with a less-than sign when it compares the widths:

```
// --snip--
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width < other.width && self.height > other.height
    }
}
```

Running the tests now produces the following:

```
$ cargo test
   Compiling rectangle v0.1.0 (file:///projects/rectangle)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/rectangle-6584c4561e48942e)

running 2 tests
test tests::larger_can_hold_smaller ... FAILED
test tests::smaller_cannot_hold_larger ... ok

failures:

---- tests::larger_can_hold_smaller stdout ----
thread 'main' panicked at 'assertion failed: larger.can_hold(&smaller)',
src/lib.rs:28:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Our tests caught the bug! Because `larger.width` is 8 and `smaller.width` is 5, the comparison of the widths in `can_hold` now returns `false`: 8 is not less than 5.

Testing Equality with the `assert_eq!` and `assert_ne!` Macros

A common way to verify functionality is to test for equality between the result of the code under test and the value you expect the code to return. You could do this using the `assert!` macro and passing it an expression using the `==` operator. However, this is such a common test that the standard library provides a pair of macros— `assert_eq!` and `assert_ne!` —to perform this test

more conveniently. These macros compare two arguments for equality or inequality, respectively. They'll also print the two values if the assertion fails, which makes it easier to see *why* the test failed; conversely, the `assert!` macro only indicates that it got a `false` value for the `==` expression, without printing the values that led to the `false` value.

In Listing 11-7, we write a function named `add_two` that adds `2` to its parameter, then we test this function using the `assert_eq!` macro.

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

Listing 11-7: Testing the function `add_two` using the `assert_eq!` macro

Let's check that it passes!

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

We pass `4` as the argument to `assert_eq!`, which is equal to the result of calling `add_two(2)`. The line for this test is `test tests::it_adds_two ... ok`, and the `ok` text indicates that our test passed!

Let's introduce a bug into our code to see what `assert_eq!` looks like when it fails. Change the implementation of the `add_two` function to instead add `3`:

```
pub fn add_two(a: i32) -> i32 {
    a + 3
}
```

Run the tests again:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `4`,
 right: `5`', src/lib.rs:11:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Our test caught the bug! The `it_adds_two` test failed, and the message tells us that the assertion that fails was `assertion failed: `(left == right)`` and what the `left` and `right` values are. This message helps us start debugging: the `left` argument was `4` but the `right` argument, where we had `add_two(2)`, was `5`. You can imagine that this would be especially helpful when we have a lot of tests going on.

Note that in some languages and test frameworks, the parameters to equality assertion functions are called `expected` and `actual`, and the order in which we specify the arguments matters. However, in Rust, they're called `left` and `right`, and the order in which we specify the value we expect and the value the code produces doesn't matter. We could write the assertion in this test as

`assert_eq!(add_two(2), 4)`, which would result in the same failure message that displays `assertion failed: `(left == right)``.

The `assert_ne!` macro will pass if the two values we give it are not equal and fail if they're equal. This macro is most useful for cases when we're not sure what a value *will* be, but we know what the value definitely *shouldn't* be. For example, if we're testing a function that is guaranteed to change its input in some way, but the way in which the input is changed depends on the day of the week that we run our tests, the best thing to assert might be that the output of the function is not equal to the input.

Under the surface, the `assert_eq!` and `assert_ne!` macros use the operators `==` and `!=`, respectively. When the assertions fail, these macros print their arguments using debug formatting, which means the values being compared must implement the `PartialEq` and `Debug` traits. All primitive types and most of the standard library types implement these traits. For structs and enums that you define yourself, you'll need to implement `PartialEq` to assert equality of those types. You'll also need to implement `Debug` to print the values when the assertion fails. Because both traits are derivable traits, as mentioned in Listing 5-12 in Chapter 5, this is usually as straightforward as adding the `#[derive(PartialEq, Debug)]` annotation to your struct or enum definition. See Appendix C, "Derivable Traits," for more details about these and other derivable traits.

Adding Custom Failure Messages

You can also add a custom message to be printed with the failure message as optional arguments to the `assert!`, `assert_eq!`, and `assert_ne!` macros. Any arguments specified after the required arguments are passed along to the `format!` macro (discussed in Chapter 8 in the "Concatenation with the `+` Operator or the `format!` Macro" section), so you can pass a format string that contains `{}` placeholders and values to go in those placeholders. Custom messages are useful for documenting what an assertion means; when a test fails, you'll have a better idea of what the problem is with the code.

For example, let's say we have a function that greets people by name and we want to test that the name we pass into the function appears in the output:

Filename: src/lib.rs

```
pub fn greeting(name: &str) -> String {
    format!("Hello {}!", name)
}

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

    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(result.contains("Carol"));
    }
}
```

The requirements for this program haven't been agreed upon yet, and we're pretty sure the `Hello` text at the beginning of the greeting will change. We decided we don't want to have to update the test when the requirements change, so instead of checking for exact equality to the value returned from the `greeting` function, we'll just assert that the output contains the text of the input parameter.

Now let's introduce a bug into this code by changing `greeting` to exclude `name` to see what the default test failure looks like:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

Running this test produces the following:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'assertion failed: result.contains(\"Carol\")',
src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

This result just indicates that the assertion failed and which line the assertion is on. A more useful failure message would print the value from the `greeting` function. Let's add a custom failure message composed of a format string with a placeholder filled in with the actual value we got from the `greeting` function:

```
    #[test]
    fn greeting_contains_name() {
        let result = greeting("Carol");
        assert!(
            result.contains("Carol"),
            "Greeting did not contain name, value was `{}`",
            result
        );
    }
```

Now when we run the test, we'll get a more informative error message:

```
$ cargo test
   Compiling greeter v0.1.0 (file:///projects/greeter)
    Finished test [unoptimized + debuginfo] target(s) in 0.93s
     Running unittests (target/debug/deps/greeter-170b942eb5bf5e3a)

running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
thread 'main' panicked at 'Greeting did not contain name, value was `Hello!`',
src/lib.rs:12:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::greeting_contains_name

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

We can see the value we actually got in the test output, which would help us debug what happened instead of what we were expecting to happen.

Checking for Panics with `should_panic`

In addition to checking return values, it's important to check that our code handles error conditions as we expect. For example, consider the `Guess` type that we created in Chapter 9, Listing 9-13. Other code that uses `Guess` depends on the guarantee that `Guess` instances will contain only values between 1 and 100. We can write a test that ensures that attempting to create a `Guess` instance with a value outside that range panics.

We do this by adding the attribute `should_panic` to our test function. The test passes if the code inside the function panics; the test fails if the code inside the function doesn't panic.

Listing 11-8 shows a test that checks that the error conditions of `Guess::new` happen when we expect them to.

Filename: src/lib.rs

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-8: Testing that a condition will cause a `panic!`

We place the `#[should_panic]` attribute after the `#[test]` attribute and before the test function it applies to. Let's look at the result when this test passes:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests guessing_game

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Looks good! Now let's introduce a bug in our code by removing the condition that the `new` function will panic if the value is greater than 100:

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }
}
```

When we run the test in Listing 11-8, it will fail:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
note: test did not panic as expected

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

We don't get a very helpful message in this case, but when we look at the test function, we see that it's annotated with `#[should_panic]`. The failure we got means that the code in the test function did not cause a panic.

Tests that use `should_panic` can be imprecise. A `should_panic` test would pass even if the test panics for a different reason from the one we were expecting. To make `should_panic` tests more precise, we can add an optional `expected` parameter to the `should_panic` attribute. The test harness will make sure that the failure message contains the provided text. For example, consider the modified code for `Guess` in Listing 11-9 where the `new` function panics with different messages depending on whether the value is too small or too large.

Filename: src/lib.rs

```
// --snip--
impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        }

        Guess { value }
    }
}

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

    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100() {
        Guess::new(200);
    }
}
```

Listing 11-9: Testing for a `panic!` with a particular panic message

This test will pass because the value we put in the `should_panic` attribute's `expected` parameter is a substring of the message that the `Guess::new` function panics with. We could have specified the entire panic message that we expect, which in this case would be `Guess value must be less than or equal to 100, got 200.` What you choose to specify depends on how much of the panic message is unique or dynamic and how precise you want your test to be. In this case, a substring of

the panic message is enough to ensure that the code in the test function executes the `else if value > 100` case.

To see what happens when a `should_panic` test with an `expected` message fails, let's again introduce a bug into our code by swapping the bodies of the `if value < 1` and the `else if value > 100` blocks:

```
        if value < 1 {
            panic!(
                "Guess value must be less than or equal to 100, got {}.",
                value
            );
        } else if value > 100 {
            panic!(
                "Guess value must be greater than or equal to 1, got {}.",
                value
            );
        }
```

This time when we run the `should_panic` test, it will fail:

```
$ cargo test
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished test [unoptimized + debuginfo] target(s) in 0.66s
     Running unittests (target/debug/deps/guessing_game-57d70c3acb738f4d)

running 1 test
test tests::greater_than_100 - should panic ... FAILED

failures:

---- tests::greater_than_100 stdout ----
thread 'main' panicked at 'Guess value must be greater than or equal to 1, got 200.',
src/lib.rs:13:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
note: panic did not contain expected string
      panic message: `"Guess value must be greater than or equal to 1, got 200."`,
 expected substring: `"Guess value must be less than or equal to 100"`

failures:
    tests::greater_than_100

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

The failure message indicates that this test did indeed panic as we expected, but the panic message did not include the expected string `'Guess value must be less than or equal to 100'`. The panic message that we did get in this case was `Guess value must be greater than or equal to 1, got 200.` Now we can start figuring out where our bug is!

Using `Result<T, E>` in Tests

Our tests so far all panic when they fail. We can also write tests that use `Result<T, E>`! Here's the test from Listing 11-1, rewritten to use `Result<T, E>` and return an `Err` instead of panicking:

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String> {
        if 2 + 2 == 4 {
            Ok(())
        } else {
            Err(String::from("two plus two does not equal four"))
        }
    }
}
```

The `it_works` function now has the `Result<(), String>` return type. In the body of the function, rather than calling the `assert_eq!` macro, we return `Ok(())` when the test passes and an `Err` with a `String` inside when the test fails.

Writing tests so they return a `Result<T, E>` enables you to use the question mark operator in the body of tests, which can be a convenient way to write tests that should fail if any operation within them returns an `Err` variant.

You can't use the `#[should_panic]` annotation on tests that use `Result<T, E>`. To assert that an operation returns an `Err` variant, *don't* use the question mark operator on the `Result<T, E>` value. Instead, use `assert!(value.is_err())`.

Now that you know several ways to write tests, let's look at what is happening when we run our tests and explore the different options we can use with `cargo test`.

Controlling How Tests Are Run

Just as `cargo run` compiles your code and then runs the resulting binary, `cargo test` compiles your code in test mode and runs the resulting test binary. The default behavior of the binary produced by `cargo test` is to run all the tests in parallel and capture output generated during test runs, preventing the output from being displayed and making it easier to read the output related to the test results. You can, however, specify command line options to change this default behavior.

Some command line options go to `cargo test`, and some go to the resulting test binary. To separate these two types of arguments, you list the arguments that go to `cargo test` followed by the separator `--` and then the ones that go to the test binary. Running `cargo test --help` displays the options you can use with `cargo test`, and running `cargo test -- --help` displays the options you can use after the separator.

Running Tests in Parallel or Consecutively

When you run multiple tests, by default they run in parallel using threads, meaning they finish running faster and you get feedback quicker. Because the tests are running at the same time, you must make sure your tests don't depend on each other or on any shared state, including a shared environment, such as the current working directory or environment variables.

For example, say each of your tests runs some code that creates a file on disk named *test-output.txt* and writes some data to that file. Then each test reads the data in that file and asserts that the file contains a particular value, which is different in each test. Because the tests run at the same time, one test might overwrite the file in the time between another test writing and reading the file. The second test will then fail, not because the code is incorrect but because the tests have interfered with each other while running in parallel. One solution is to make sure each test writes to a different file; another solution is to run the tests one at a time.

If you don't want to run the tests in parallel or if you want more fine-grained control over the number of threads used, you can send the `--test-threads` flag and the number of threads you want to use to the test binary. Take a look at the following example:

```
$ cargo test -- --test-threads=1
```

We set the number of test threads to `1`, telling the program not to use any parallelism. Running the tests using one thread will take longer than running them in parallel, but the tests won't interfere with each other if they share state.

Showing Function Output

By default, if a test passes, Rust's test library captures anything printed to standard output. For example, if we call `println!` in a test and the test passes, we won't see the `println!` output in the terminal; we'll see only the line that indicates the test passed. If a test fails, we'll see whatever was printed to standard output with the rest of the failure message.

As an example, Listing 11-10 has a silly function that prints the value of its parameter and returns 10, as well as a test that passes and a test that fails.

Filename: src/lib.rs

```
fn prints_and_returns_10(a: i32) -> i32 {
    println!("I got the value {}", a);
    10
}

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

    #[test]
    fn this_test_will_pass() {
        let value = prints_and_returns_10(4);
        assert_eq!(10, value);
    }

    #[test]
    fn this_test_will_fail() {
        let value = prints_and_returns_10(8);
        assert_eq!(5, value);
    }
}
```

Listing 11-10: Tests for a function that calls `println!`

When we run these tests with `cargo test`, we'll see the following output:

```
$ cargo test
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running unittests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Note that nowhere in this output do we see `I got the value 4`, which is what is printed when the test that passes runs. That output has been captured. The output from the test that failed, `I got the value 8`, appears in the section of the test summary output, which also shows the cause of the test failure.

If we want to see printed values for passing tests as well, we can tell Rust to also show the output of successful tests at the end with `--show-output`.

```
$ cargo test -- --show-output
```

When we run the tests in Listing 11-10 again with the `--show-output` flag, we see the following output:

```
$ cargo test -- --show-output
   Compiling silly-function v0.1.0 (file:///projects/silly-function)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests (target/debug/deps/silly_function-160869f38cff9166)

running 2 tests
test tests::this_test_will_fail ... FAILED
test tests::this_test_will_pass ... ok

successes:

---- tests::this_test_will_pass stdout ----
I got the value 4


successes:
    tests::this_test_will_pass

failures:

---- tests::this_test_will_fail stdout ----
I got the value 8
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `5`,
 right: `10`', src/lib.rs:19:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

error: test failed, to rerun pass '--lib'
```

Running a Subset of Tests by Name

Sometimes, running a full test suite can take a long time. If you're working on code in a particular area, you might want to run only the tests pertaining to that code. You can choose which tests to run by passing `cargo test` the name or names of the test(s) you want to run as an argument.

To demonstrate how to run a subset of tests, we'll first create three tests for our `add_two` function, as shown in Listing 11-11, and choose which ones to run.

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn add_two_and_two() {
        assert_eq!(4, add_two(2));
    }

    #[test]
    fn add_three_and_two() {
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred() {
        assert_eq!(102, add_two(100));
    }
}
```

Listing 11-11: Three tests with three different names

If we run the tests without passing any arguments, as we saw earlier, all the tests will run in parallel:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.62s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 3 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Running Single Tests

We can pass the name of any test function to `cargo test` to run only that test:

```
$ cargo test one_hundred
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.69s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in
0.00s
```

Only the test with the name `one_hundred` ran; the other two tests didn't match that name. The test output lets us know we had more tests that didn't run by displaying `2 filtered out` at the end.

We can't specify the names of multiple tests in this way; only the first value given to `cargo test` will be used. But there is a way to run multiple tests.

Filtering to Run Multiple Tests

We can specify part of a test name, and any test whose name matches that value will be run. For example, because two of our tests' names contain `add`, we can run those two by running `cargo test add`:

```
$ cargo test add
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test tests::add_three_and_two ... ok
test tests::add_two_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in
0.00s
```

This command ran all tests with `add` in the name and filtered out the test named `one_hundred`. Also note that the module in which a test appears becomes part of the test's name, so we can run all the tests in a module by filtering on the module's name.

Ignoring Some Tests Unless Specifically Requested

Sometimes a few specific tests can be very time-consuming to execute, so you might want to exclude them during most runs of `cargo test`. Rather than listing as arguments all tests you do want to run, you can instead annotate the time-consuming tests using the `ignore` attribute to exclude them, as shown here:

Filename: src/lib.rs

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

After `#[test]` we add the `#[ignore]` line to the test we want to exclude. Now when we run our tests, `it_works` runs, but `expensive_test` doesn't:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.60s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

The `expensive_test` function is listed as `ignored`. If we want to run only the ignored tests, we can use `cargo test -- --ignored`:

```
$ cargo test -- --ignored
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.61s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

By controlling which tests run, you can make sure your `cargo test` results will be fast. When you're at a point where it makes sense to check the results of the `ignored` tests and you have time to wait for the results, you can run `cargo test -- --ignored` instead. If you want to run all tests whether they're ignored or not, you can run `cargo test -- --include-ignored`.

Test Organization

As mentioned at the start of the chapter, testing is a complex discipline, and different people use different terminology and organization. The Rust community thinks about tests in terms of two main categories: unit tests and integration tests. *Unit tests* are small and more focused, testing one module in isolation at a time, and can test private interfaces. *Integration tests* are entirely external to your library and use your code in the same way any other external code would, using only the public interface and potentially exercising multiple modules per test.

Writing both kinds of tests is important to ensure that the pieces of your library are doing what you expect them to, separately and together.

Unit Tests

The purpose of unit tests is to test each unit of code in isolation from the rest of the code to quickly pinpoint where code is and isn't working as expected. You'll put unit tests in the *src* directory in each file with the code that they're testing. The convention is to create a module named `tests` in each file to contain the test functions and to annotate the module with `cfg(test)`.

The Tests Module and `#[cfg(test)]`

The `#[cfg(test)]` annotation on the tests module tells Rust to compile and run the test code only when you run `cargo test`, not when you run `cargo build`. This saves compile time when you only want to build the library and saves space in the resulting compiled artifact because the tests are not included. You'll see that because integration tests go in a different directory, they don't need the `#[cfg(test)]` annotation. However, because unit tests go in the same files as the code, you'll use `#[cfg(test)]` to specify that they shouldn't be included in the compiled result.

Recall that when we generated the new `adder` project in the first section of this chapter, Cargo generated this code for us:

Filename: src/lib.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        let result = 2 + 2;
        assert_eq!(result, 4);
    }
}
```

This code is the automatically generated test module. The attribute `cfg` stands for *configuration* and tells Rust that the following item should only be included given a certain configuration option. In this case, the configuration option is `test`, which is provided by Rust for compiling and running tests. By using the `cfg` attribute, Cargo compiles our test code only if we actively run the tests with `cargo test`. This includes any helper functions that might be within this module, in addition to the functions annotated with `#[test]`.

Testing Private Functions

There's debate within the testing community about whether or not private functions should be tested directly, and other languages make it difficult or impossible to test private functions. Regardless of which testing ideology you adhere to, Rust's privacy rules do allow you to test private functions. Consider the code in Listing 11-12 with the private function `internal_adder`.

Filename: src/lib.rs

```
pub fn add_two(a: i32) -> i32 {
    internal_adder(a, 2)
}

fn internal_adder(a: i32, b: i32) -> i32 {
    a + b
}

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

    #[test]
    fn internal() {
        assert_eq!(4, internal_adder(2, 2));
    }
}
```

Listing 11-12: Testing a private function

Note that the `internal_adder` function is not marked as `pub`. Tests are just Rust code, and the `tests` module is just another module. As we discussed in the "Paths for Referring to an Item in the Module Tree" section, items in child modules can use the items in their ancestor modules. In this test, we bring all of the `test` module's parent's items into scope with `use super::*`, and then the test can call `internal_adder`. If you don't think private functions should be tested, there's nothing in Rust that will compel you to do so.

Integration Tests

In Rust, integration tests are entirely external to your library. They use your library in the same way any other code would, which means they can only call functions that are part of your library's public API. Their purpose is to test whether many parts of your library work together correctly. Units of

code that work correctly on their own could have problems when integrated, so test coverage of the integrated code is important as well. To create integration tests, you first need a *tests* directory.

The *tests* Directory

We create a *tests* directory at the top level of our project directory, next to *src*. Cargo knows to look for integration test files in this directory. We can then make as many test files as we want, and Cargo will compile each of the files as an individual crate.

Let's create an integration test. With the code in Listing 11-12 still in the *src/lib.rs* file, make a *tests* directory, create a new file named *tests/integration_test.rs*, and enter the code in Listing 11-13.

Filename: tests/integration_test.rs

```
use adder;

#[test]
fn it_adds_two() {
    assert_eq!(4, adder::add_two(2));
}
```

Listing 11-13: An integration test of a function in the `adder` crate

Each file in the `tests` directory is a separate crate, so we need to bring our library into each test crate's scope. For that reason we add `use adder` at the top of the code, which we didn't need in the unit tests.

We don't need to annotate any code in *tests/integration_test.rs* with `#[cfg(test)]`. Cargo treats the `tests` directory specially and compiles files in this directory only when we run `cargo test`. Run `cargo test` now:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 1.31s
     Running unittests (target/debug/deps/adder-1082c4b063a8fbe6)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-
1082c4b063a8fbe6)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

The three sections of output include the unit tests, the integration test, and the doc tests. The first section for the unit tests is the same as we've been seeing: one line for each unit test (one named `internal` that we added in Listing 11-12) and then a summary line for the unit tests.

The integration tests section starts with the line `Running tests/integration_test.rs`. Next, there is a line for each test function in that integration test and a summary line for the results of the integration test just before the `Doc-tests adder` section starts.

Each integration test file has its own section, so if we add more files in the *tests* directory, there will be more integration test sections.

We can still run a particular integration test function by specifying the test function's name as an argument to `cargo test`. To run all the tests in a particular integration test file, use the `--test` argument of `cargo test` followed by the name of the file:

```
$ cargo test --test integration_test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.64s
     Running tests/integration_test.rs (target/debug/deps/integration_test-
82e7799c1bc62298)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

This command runs only the tests in the *tests/integration_test.rs* file.

Submodules in Integration Tests

As you add more integration tests, you might want to make more files in the *tests* directory to help organize them; for example, you can group the test functions by the functionality they're testing. As mentioned earlier, each file in the *tests* directory is compiled as its own separate crate, which is useful for creating separate scopes to more closely imitate the way end users will be using your crate. However, this means files in the *tests* directory don't share the same behavior as files in *src* do, as you learned in Chapter 7 regarding how to separate code into modules and files.

The different behavior of *tests* directory files is most noticeable when you have a set of helper functions to use in multiple integration test files and you try to follow the steps in the "Separating

Modules into Different Files" section of Chapter 7 to extract them into a common module. For example, if we create *tests/common.rs* and place a function named `setup` in it, we can add some code to `setup` that we want to call from multiple test functions in multiple test files:

Filename: tests/common.rs

```
pub fn setup() {
    // setup code specific to your library's tests would go here
}
```

When we run the tests again, we'll see a new section in the test output for the *common.rs* file, even though this file doesn't contain any test functions nor did we call the `setup` function from anywhere:

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.89s
     Running unittests (target/debug/deps/adder-92948b65e88960b4)

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

     Running tests/common.rs (target/debug/deps/common-92948b65e88960b4)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

     Running tests/integration_test.rs (target/debug/deps/integration_test-
92948b65e88960b4)

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

Having `common` appear in the test results with `running 0 tests` displayed for it is not what we wanted. We just wanted to share some code with the other integration test files.

To avoid having `common` appear in the test output, instead of creating *tests/common.rs*, we'll create *tests/common/mod.rs*. This is an alternate naming convention that Rust also understands. Naming the file this way tells Rust not to treat the `common` module as an integration test file. When we move the `setup` function code into *tests/common/mod.rs* and delete the *tests/common.rs* file, the section in the test output will no longer appear. Files in subdirectories of the *tests* directory don't get compiled as separate crates or have sections in the test output.

After we've created *tests/common/mod.rs*, we can use it from any of the integration test files as a module. Here's an example of calling the `setup` function from the `it_adds_two` test in *tests/integration_test.rs*:

Filename: tests/integration_test.rs

```
use adder;

mod common;

#[test]
fn it_adds_two() {
    common::setup();
    assert_eq!(4, adder::add_two(2));
}
```

Note that the `mod common;` declaration is the same as the module declaration we demonstrated in Listing 7-21. Then in the test function, we can call the `common::setup()` function.

Integration Tests for Binary Crates

If our project is a binary crate that only contains a *src/main.rs* file and doesn't have a *src/lib.rs* file, we can't create integration tests in the *tests* directory and bring functions defined in the *src/main.rs* file into scope with a `use` statement. Only library crates expose functions that other crates can use; binary crates are meant to be run on their own.

This is one of the reasons Rust projects that provide a binary have a straightforward *src/main.rs* file that calls logic that lives in the *src/lib.rs* file. Using that structure, integration tests *can* test the library crate with `use` to make the important functionality available. If the important functionality works, the small amount of code in the *src/main.rs* file will work as well, and that small amount of code doesn't need to be tested.

Summary

Rust's testing features provide a way to specify how code should function to ensure it continues to work as you expect, even as you make changes. Unit tests exercise different parts of a library separately and can test private implementation details. Integration tests check that many parts of the library work together correctly, and they use the library's public API to test the code in the same way external code will use it. Even though Rust's type system and ownership rules help prevent some kinds of bugs, tests are still important to reduce logic bugs having to do with how your code is expected to behave.

Let's combine the knowledge you learned in this chapter and in previous chapters to work on a project!