

Foreword

It wasn't always so clear, but the Rust programming language is fundamentally different. No matter what kind of code you are writing now, Rust empowers you to reanalyze your code with confidence in a wider variety of domains than you did before.

Take, for example, "systems-level" work that deals with low-level details of memory management, data representation, and concurrency. Traditionally, this realm of programming has been accessible only to a select few who have devoted the necessary years learning and practicing. And even those who practice it do so with caution, lest their code be prone to crashes, or corruption.

Rust breaks down these barriers by eliminating the old pitfalls and providing a range of tools to help you along the way. Programmers who need to "dip down" into systems-level code can do so with Rust, without taking on the customary risk of crashes or security vulnerabilities. By having to learn the fine points of a fickle toolchain, Better yet, the language is designed to point naturally towards reliable code that is efficient in terms of speed and memory usage.

Programmers who are already working with low-level code can use Rust to reduce the risk. For example, introducing parallelism in Rust is a relatively low-risk operation: the language prevents many of the classical mistakes for you. And you can tackle more aggressive optimizations with confidence that you won't accidentally introduce crashes or exploits.

But Rust isn't limited to low-level systems programming. It's expressive and flexible enough to make CLI apps, web servers, and many other kinds of code quite pleasant to write. See the simple examples of both later in the book. Working with Rust allows you to learn how to move from one domain to another; you can learn Rust by writing a web app, then use the same skills to target your Raspberry Pi.

This book fully embraces the potential of Rust to empower its users. It's a friendly introduction to the language, intended to help you level up not just your knowledge of Rust, but also your confidence as a programmer in general. So dive in, get ready to learn—and have fun!

— Nicholas Matsakis and Aaron Turon

Introduction

Note: This edition of the book is the same as [The Rust Programming Language](#), available in print and ebook format from [No Starch Press](#).

Welcome to *The Rust Programming Language*, an introductory book about Rust. The book helps you write faster, more reliable software. High-level and low-level control are often at odds in programming language design; Rust changes that dynamic. Through balancing powerful technical capacity and a great developer experience, Rust gives you the option to control low-level details (such as memory usage) without all the headache associated with such control.

Who Rust Is For

Rust is ideal for many people for a variety of reasons. Let's look at a few of the most common groups.

Teams of Developers

Rust is proving to be a productive tool for collaborating among large teams with varying levels of systems programming knowledge. Low-level code is prone to bugs, which in most other languages can be caught only through extensive testing and review by experienced developers. In Rust, the compiler plays a gatekeeper role, catching bugs in the compile code with these elusive bugs, including concurrency bugs. By working with the compiler, the team can spend their time focusing on the program's logic rather than catching bugs.

Rust also brings contemporary developer tools to the systems programming ecosystem:

- Cargo, the included dependency manager and build tool, makes adding and managing dependencies painless and consistent across the Rust ecosystem.
- Rustfmt ensures a consistent coding style across developers.
- The Rust Language Server powers Integrated Development Environments with code completion and inline error messages.

By using these and other tools in the Rust ecosystem, developers can be productive at the systems-level code.

Students

Rust is for students and those who are interested in learning about systems programming. Many people have learned about topics like operating systems development and networking, and are welcoming and happy to answer student questions. Through efforts such as the Rust Foundation, teams want to make systems concepts more accessible to more people, especially students of programming.

Companies

Hundreds of companies, large and small, use Rust in production for a variety of projects. Companies include command line tools, web services, DevOps tooling, embedded devices, mobile applications, data analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even major parts of the Firefox web browser.

Open Source Developers

Rust is for people who want to build the Rust programming language, community, and libraries. We'd love to have you contribute to the Rust language.

People Who Value Speed and Stability

Rust is for people who crave speed and stability in a language. By speed, we mean programs that you can create with Rust and the speed at which Rust lets you write them. By stability, we mean that the compiler's checks ensure stability through feature additions and refactoring. Most other languages have trouble with brittle legacy code in languages without these checks, which developers are used to. By striving for zero-cost abstractions, higher-level features that compile to low-level code written manually, Rust endeavors to make safe code be fast code as well.

The Rust language hopes to support many other users as well; those mentioned here are just some of the biggest stakeholders. Overall, Rust's greatest ambition is to eliminate the trade-offs that programmers have accepted for decades by providing safety *and* productivity.

ergonomics. Give Rust a try and see if its choices work for you.

Who This Book Is For

This book assumes that you've written code in another programming language at some point, and it makes certain assumptions about which one. We've tried to make the material broadly accessible to people with a wide variety of programming backgrounds. We don't spend a lot of time talking about what programming *is* or how to think about it. If you're entirely new to programming, we recommend that you start by reading a book that specifically provides an introduction to programming.

How to Use This Book

In general, this book assumes that you're reading it in sequence from front to back. You'll build on concepts in earlier chapters, and earlier chapters might not delve into every topic in great detail. You might need to revisit the topic in a later chapter.

You'll find two kinds of chapters in this book: concept chapters and project chapters. In concept chapters, you'll learn about an aspect of Rust. In project chapters, we'll build a small program together, applying what you've learned so far. Chapters 2, 12, and 20 are project chapters; the other chapters are concept chapters.

Chapter 1 explains how to install Rust, how to write a Hello, world! program, and how to use Rust's package manager and build tool. Chapter 2 is a hands-on introduction to Rust's ownership system. Here we cover concepts at a high level, and later chapters will provide additional details. If you want to get your hands dirty right away, Chapter 2 is the place for that. At first, you might skip Chapter 3, which covers Rust features similar to those of other programming languages. Instead, you might want to head straight to Chapter 4 to learn about Rust's ownership system. However, if you're a meticulous learner who prefers to learn every detail before moving on to the next chapter, you might want to skip Chapter 2 and go straight to Chapter 3, returning to Chapter 2 when you're ready to apply the details you've learned.

Chapter 5 discusses structs and methods, and Chapter 6 covers enums, match statements, and the let control flow construct. You'll use structs and enums to make custom types and functions.

In Chapter 7, you'll learn about Rust's module system and about privacy rules. Chapter 8 discusses how to use Rust's standard library to build programs that interact with the outside world. Chapter 9 explores Rust's error-handling philosophy and techniques.

Chapter 10 digs into generics, traits, and lifetimes, which give you the power to write reusable code that applies to multiple types. Chapter 11 is all about testing, which is necessary to ensure your program's logic is correct. In Chapter 12, we'll build a command-line application that uses the grep command line to search through files. For this, we'll use many of the concepts we discussed in the previous chapters.

Chapter 13 explores closures and iterators: features of Rust that come from functional programming languages. In Chapter 14, we'll examine Cargo in more depth and talk about how to share your libraries with others. Chapter 15 discusses smart pointers that provide memory safety and the traits that enable their functionality.

In Chapter 16, we'll walk through different models of concurrent programming. Chapter 17 looks at how Rust helps you to program in multiple threads fearlessly. Chapter 18 compares Rust's object-oriented programming principles to those you might be familiar with from other languages.

Chapter 18 is a reference on patterns and pattern matching, which are powerful ideas throughout Rust programs. Chapter 19 contains a smorgasbord of advanced topics including unsafe Rust and more about lifetimes, traits, types, functions, and closures.

In Chapter 20, we'll complete a project in which we'll implement a low-level reverse proxy server!

Finally, some appendixes contain useful information about the language in a compact format. Appendix A covers Rust's keywords, Appendix B covers Rust's operators, Appendix C covers derivable traits provided by the standard library, and Appendix D covers the standard library itself.

There is no wrong way to read this book: if you want to skip ahead, go for it! You can always go back to earlier chapters if you experience any confusion. But do whatever works best for you.

An important part of the process of learning Rust is learning how to read the error messages the compiler displays: these will guide you toward working code. As such, we'll provide examples of code that doesn't compile along with the error message the compiler will output. Know that if you enter and run a random example, it may not come from the surrounding text to see whether the example you're trying to run is meaningful. In such situations, we'll lead you to the correct version of any code that doesn't compile.

Source Code

The source files from which this book is generated can be found on [GitHub](#).

Getting Started

Let's start your Rust journey! There's a lot to learn, but every journey starts somewhere. In this chapter, we'll discuss:

- Installing Rust on Linux, macOS, and Windows
- Writing a program that prints `Hello, world!`
- Using `cargo`, Rust's package manager and build system

Installation

The first step is to install Rust. We'll download Rust through `rustup`, a command-line tool for managing Rust versions and associated tools. You'll need an internet connection to use `rustup`.

Note: If you prefer not to use `rustup` for some reason, please see [the Rust documentation](#) for other options.

The following steps install the latest stable version of the Rust compiler. Rust is a fast-moving project, so the steps here might change over time to ensure that all the examples in the book that compile will continue to compile in future versions. The output might differ slightly between versions, because Rust often emits different diagnostic messages and warnings. In other words, any newer, stable version of Rust you install should work as expected with the content of this book.

Command Line Notation

In this chapter and throughout the book, we'll show some commands use Lines that you should enter in a terminal all start with `$`. You don't need to type the `$` character; it indicates the start of each command. Lines that don't start with `$` are the output of the previous command. Additionally, PowerShell-specific examples will use the `PS` prompt rather than `$`.

Installing `rustup` on Linux or macOS

If you're using Linux or macOS, open a terminal and enter the following command:

```
$ curl https://sh.rustup.rs -sSf | sh
```

The command downloads a script and starts the installation of the `rustup` tool. It will download the latest stable version of Rust. You might be prompted for your password. If the command succeeds, the following line will appear:

```
Rust is installed now. Great!
```

If you prefer, feel free to download the script and inspect it before running it:

The installation script automatically adds Rust to your system PATH after you run it. To start using Rust right away instead of restarting your terminal, run the following command in a shell to add Rust to your system PATH manually:

```
$ source $HOME/.cargo/env
```

Alternatively, you can add the following line to your `~/.bash_profile`:

```
$ export PATH="$HOME/.cargo/bin:$PATH"
```

Additionally, you'll need a linker of some kind. It's likely one is already installed, but if you get errors while trying to compile a Rust program and get errors indicating that a linker could not be found, you may need to install one manually. C programs require a C compiler, and many common Rust packages depend on C code and will need a C compiler. There are several options for installing one now.

Installing `rustup` on Windows

On Windows, go to <https://www.rust-lang.org/install.html> and follow the instructions to install Rust. At some point in the installation, you'll receive a message explaining that you need to install the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire them is to download the [Build Tools for Visual Studio 2017](#). The tools are in the Other Tools and Frameworks section of the download page.

The rest of this book uses commands that work in both `cmd.exe` and PowerShell. We'll explain the differences, and you'll learn which to use.

Updating and Uninstalling

After you've installed Rust via `rustup`, updating to the latest version is easy. Run the following update script:

```
$ rustup update
```

To uninstall Rust and `rustup`, run the following uninstall script from your shell:

```
$ rustup self uninstall
```

Troubleshooting

To check whether you have Rust installed correctly, open a shell and enter the command:

```
$ rustc --version
```

You should see the version number, commit hash, and commit date for the last time Rust has been released in the following format:

```
rustc x.y.z (abcabcabc yyyy-mm-dd)
```

If you see this information, you have installed Rust successfully! If you don't see it, you're on Windows, check that Rust is in your `%PATH%` system variable. If that still isn't working, there are a number of places you can get help. The easiest way is to join the [irc.mozilla.org](#) channel, which you can access through [Mibbit](#). At that address you will find Rustaceans (a silly nickname we call ourselves) who can help you out. Other places to seek help include the [Users forum](#) and [Stack Overflow](#).

Local Documentation

The installer also includes a copy of the documentation locally, so you can run `rustup doc` to open the local documentation in your browser.

Any time a type or function is provided by the standard library and you're not sure how to use it, use the application programming interface (API) documentation.

Hello, World!

Now that you've installed Rust, let's write your first Rust program. It's traditional to start with the Hello, world! program, so that's what we'll do here!

Note: This book assumes basic familiarity with the command line. Rust makes demands about your editing or tooling or where your code lives, so if you prefer to use an integrated development environment (IDE) instead of the command line, you may want to consider using your favorite IDE. Many IDEs now have some degree of Rust support; check the [Rust ecosystem page](#) for details. Recently, the Rust team has been focusing on enabling great IDE support, and progress has been made rapidly on that front!

Creating a Project Directory

You'll start by making a directory to store your Rust code. It doesn't matter where this directory lives, but for the exercises and projects in this book, we suggest making a `projects` directory in your home directory and keeping all your projects there.

Open a terminal and enter the following commands to make a `projects` directory:

the Hello, world! project within the *projects* directory.

For Linux and macOS, enter this:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

For Windows CMD, enter this:

```
> mkdir "%USERPROFILE%\projects"
> cd /d "%USERPROFILE%\projects"
> mkdir hello_world
> cd hello_world
```

For Windows PowerShell, enter this:

```
> mkdir $env:USERPROFILE\projects
> cd $env:USERPROFILE\projects
> mkdir hello_world
> cd hello_world
```

Writing and Running a Rust Program

Next, make a new source file and call it *main.rs*. Rust files always end with `.rs`. If you’re using more than one word in your filename, use an underscore to separate them, like *hello_world.rs* rather than *helloworld.rs*.

Now open the *main.rs* file you just created and enter the code in Listing 1-1.

Filename: main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Listing 1-1: A program that prints `Hello, world!`

Save the file and go back to your terminal window. On Linux or macOS, enter the commands to compile and run the file:

```
$ rustc main.rs
$ ./main
Hello, world!
```

On Windows, enter the command `.\main.exe` instead of `./main`:

```
> rustc main.rs
> .\main.exe
Hello, world!
```

Regardless of your operating system, the string `Hello, world!` should print to the terminal. If it doesn’t, or if you don’t see this output, refer back to the “Troubleshooting” part of the Installation chapter or ask for help in the [Rust forums](#).

If `Hello, world!` did print, congratulations! You’ve officially written a Rust program. Welcome to the Rust programming language!

Anatomy of a Rust Program

Let's review in detail what just happened in your Hello, world! program. Here's the code:

```
fn main() {  
}
```

These lines define a function in Rust. The `main` function is special: it is always run in every executable Rust program. The first line declares a function named `main`, which has no parameters and returns nothing. If there were parameters, they would go in the parentheses after `main`.

Also, note that the function body is wrapped in curly brackets, `{ }` . Rust requires that you always wrap function bodies in curly braces. It's good style to place the opening curly bracket on the same line as the function declaration, adding one space in between.

At the time of this writing, an automatic formatter tool called `rustfmt` is under development. If you want to stick to a standard style across Rust projects, `rustfmt` will format your code for you. The Rust team plans to eventually include this tool with the standard Rust compiler `rustc`. So depending on when you read this book, it might already be installed on your system. Check the online documentation for more details.

Inside the `main` function is the following code:

```
println!("Hello, world!");
```

This line does all the work in this little program: it prints text to the screen. There are a few details to notice here. First, Rust style is to indent with four spaces, not a tab.

Second, `println!` calls a Rust macro. If it called a function instead, it would be `print!` (without the `!`). We'll discuss Rust macros in more detail in Appendix D. For now, just know that using a `!` means that you're calling a macro instead of a normal function.

Third, you see the `"Hello, world!"` string. We pass this string as an argument to the `println!` macro, and the string is printed to the screen.

Fourth, we end the line with a semicolon `(;)`, which indicates that this expression is the last one in the function body. The next line of code is ready to begin. Most lines of Rust code end with a semicolon.

Compiling and Running Are Separate Steps

You've just run a newly created program, so let's examine each step in the process.

Before running a Rust program, you must compile it using the Rust compiler command `rustc` and passing it the name of your source file, like this:

```
$ rustc main.rs
```

If you have a C or C++ background, you'll notice that this is similar to `gcc` or `g++`. After successfully compiling the code, Rust outputs a binary executable.

On Linux and macOS you can see the executable by entering the `ls` command, which shows the following:

```
$ ls  
main main.rs
```

With PowerShell on Windows, you can use `ls` as well, but you'll see three files:

```
> ls
```

```
Directory: Path:\to\the\project
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	6/1/2018 7:31 AM	137728	main.exe
-a---	6/1/2018 7:31 AM	1454080	main.pdb
-a---	6/1/2018 7:31 AM	14	main.rs

With CMD on Windows, you would enter the following:

```
> dir /B %= the /B option says to only show the file names =%  
main.exe  
main.pdb  
main.rs
```

This shows the source code file with the `.rs` extension, the executable file (`main` on all other platforms), and, when using CMD, a file containing debugging information with the `.pdb` extension. From here, you run the `main` or `main.exe` file, like this:

```
$ ./main # or .\main.exe on Windows
```

If `main.rs` was your Hello, world! program, this line would print `Hello, world!`.

If you're more familiar with a dynamic language, such as Ruby, Python, or Java, you might be used to compiling and running a program as separate steps. Rust is an *all-in-one* language, meaning you can compile a program and give the executable to someone else who can run it even without having Rust installed. If you give someone a `.rb`, `.py`, or `.js` file, they will need to have a Ruby, Python, or JavaScript implementation installed (respectively). But with Rust, you only need one command to compile and run your program. Everything is handled for you by the compiler.

Just compiling with `rustc` is fine for simple programs, but as your project grows, you'll want a better way to manage all the options and make it easy to share your code. Next, we'll introduce `Cargo`, a build tool, which will help you write real-world Rust programs.

Hello, Cargo!

Cargo is Rust's build system and package manager. Most Rustaceans use this tool because it handles a lot of tasks for you, such as building your code, finding the libraries your code depends on, and building those libraries. (We call libraries *dependencies*.)

The simplest Rust programs, like the one we've written so far, don't have any dependencies. If you had built the Hello, world! project with Cargo, it would only use the part of `Cargo.toml` that specifies the code you wrote. As you write more complex Rust programs, you'll add dependencies to them. Once you start a project using Cargo, adding dependencies will be much easier to do.

Because the vast majority of Rust projects use Cargo, the rest of this book assumes you'll be using it. If you're using another build tool, like `rustup`, Cargo comes installed with Rust if you used the official installers (Windows, macOS, Linux).

"Installation" section. If you installed Rust through some other means, check installed by entering the following into your terminal:

```
$ cargo --version
```

If you see a version number, you have it! If you see an error, such as `command not found`, consult the documentation for your method of installation to determine how to install C

Creating a Project with Cargo

Let's create a new project using Cargo and look at how it differs from our original project. Navigate back to your `projects` directory (or wherever you decided to create your project), and on any operating system, run the following:

```
$ cargo new hello_cargo
$ cd hello_cargo
```

The first command creates a new directory called `hello_cargo`. We've named it `hello_cargo` because Cargo creates its files in a directory of the same name.

Go into the `hello_cargo` directory and list the files. You'll see that Cargo has given us a lot of help: it has created a `Cargo.toml` file and a `src` directory with a `main.rs` file in it, and initialized a new Git repository along with a `.gitignore` file.

Note: Git is a common version control system. You can change the `cargo new` command to use a different version control system or no version control system by using the `--vcs` flag. Run `cargo new --help` to see the available options.

Open `Cargo.toml` in your text editor of choice. It should look similar to the following:

Filename: `Cargo.toml`

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

Listing 1-2: Contents of `Cargo.toml` generated by `cargo new`

This file is in the [TOML](#) (*Tom's Obvious, Minimal Language*) format, which is a simple key-value pair format.

The first line, `[package]`, is a section heading that indicates that the following configuration information applies to the entire package. As we add more information to this file, we'll add other sections.

The next three lines set the configuration information Cargo needs to compile the package: the name, the version, and who wrote it. Cargo gets your name and email from your operating system's environment, so if that information is not correct, fix the information now and re-run `cargo new`.

The last line, `[dependencies]`, is the start of a section for you to list any of your dependencies. In Rust, packages of code are referred to as *crates*. We won't be adding any dependencies for this project, but we will in the first project in Chapter 2, so we'll use this definition of `[dependencies]` for now.

Now open `src/main.rs` and take a look:

Filename: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Cargo has generated a Hello, world! program for you, just like the one we wrote in the previous project. The differences between our previous project and the project Cargo generates are that the code is in the `src` directory, and we have a `Cargo.toml` configuration file in the root of the project.

Cargo expects your source files to live inside the `src` directory. The top-level project directory is for README files, license information, configuration files, and anything else you might need. Using Cargo helps you organize your projects. There's a place for everything, and a place for nothing.

If you started a project that doesn't use Cargo, as we did with the Hello, world! program, you can convert it to a project that does use Cargo. Move the project code into the `src` directory and add an appropriate `Cargo.toml` file.

Building and Running a Cargo Project

Now let's look at what's different when we build and run the Hello, world! program. First, move into your `hello_cargo` directory, build your project by entering the following command:

```
$ cargo build
Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

This command creates an executable file in `target/debug/hello_cargo` (or `target\debug\hello_cargo` on Windows) rather than in your current directory. You can run the executable by running:

```
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe
Hello, world!
```

If all goes well, `Hello, world!` should print to the terminal. Running `cargo build` also causes Cargo to create a new file at the top level: `Cargo.lock`. This file keeps track of the versions of dependencies in your project. This project doesn't have dependencies, so the file is sparse. You won't ever need to change this file manually; Cargo manages its dependencies for you.

We just built a project with `cargo build` and ran it with `./target/debug/hello_cargo`. You can also use `cargo run` to compile the code and then run the resulting executable:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/hello_cargo`
Hello, world!
```

Notice that this time we didn't see output indicating that Cargo was compiling the code. That's because Cargo figured out that the files hadn't changed, so it just ran the binary. If you had made changes to the code, Cargo would have rebuilt the project before running it, and you would have seen the compilation output.

```
$ cargo run
    Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
        Running `target/debug/hello_cargo`
Hello, world!
```

Cargo also provides a command called `cargo check`. This command quickly checks to make sure it compiles but doesn't produce an executable:

```
$ cargo check
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
   Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

Why would you not want an executable? Often, `cargo check` is much faster because it skips the step of producing an executable. If you're continually changing the code, using `cargo check` will speed up the process! As such, many developers use `cargo check` periodically as they write their program to make sure it compiles correctly before running `cargo build` when they're ready to use the executable.

Let's recap what we've learned so far about Cargo:

- We can build a project using `cargo build` or `cargo check`.
- We can build and run a project in one step using `cargo run`.
- Instead of saving the result of the build in the same directory as our code, we can save it to a `target/debug` directory.

An additional advantage of using Cargo is that the commands are the same regardless of the operating system you're working on. So, at this point, we'll no longer provide separate sections for Linux and macOS versus Windows.

Building for Release

When your project is finally ready for release, you can use `cargo build --release` with optimizations. This command will create an executable in `target/release`. The optimizations make your Rust code run faster, but turning them on lengthens the time it takes for your program to compile. This is why there are two different profiles: one for development where you want to rebuild quickly and often, and another for building the final product that won't be rebuilt repeatedly and that will run as fast as possible. If you're concerned about your code's running time, be sure to run `cargo build --release` and benchmark the resulting executable in `target/release`.

Cargo as Convention

With simple projects, Cargo doesn't provide a lot of value over just using `rustc` to compile your code. However, as your programs become more intricate, Cargo becomes increasingly useful. With complex projects containing many files and dependencies, it's much easier to let Cargo coordinate the build.

Even though the `hello_cargo` project is simple, it now uses much of the recommended best practices for building Rust projects. In fact, to work on any existing projects, you can use the following steps to clone the repository, change to its directory, and build it.

```
$ git clone someurl.com/someproject
$ cd someproject
$ cargo build
```

For more information about Cargo, check out [its documentation](#).

Summary

You're already off to a great start on your Rust journey! In this chapter, you've learned how to:

- Install the latest stable version of Rust using `rustup`
- Update to a newer Rust version

- Open locally installed documentation
- Write and run a Hello, world! program using `rustc` directly
- Create and run a new project using the conventions of Cargo

This is a great time to build a more substantial program to get used to reading code. So, in Chapter 2, we'll build a guessing game program. If you would rather learn how common programming concepts work in Rust, see Chapter 3 and then return here.

Programming a Guessing Game

Let's jump into Rust by working through a hands-on project together! This chapter will introduce you to a few common Rust concepts by showing you how to use them in a real project. `let`, `match`, methods, associated functions, using external crates, and more. You will explore these ideas in more detail. In this chapter, you'll practice the fun of writing code.

We'll implement a classic beginner programming problem: a guessing game. The program will generate a random integer between 1 and 100. It will then prompt the user for a guess. After a guess is entered, the program will indicate whether the guess is correct, the game will print a congratulatory message and exit.

Setting Up a New Project

To set up a new project, go to the `projects` directory that you created in Chapter 1 and create a new project using Cargo, like so:

```
$ cargo new guessing_game  
$ cd guessing_game
```

The first command, `cargo new`, takes the name of the project (`guessing_game`) as an argument. The second command changes to the new project's directory.

Look at the generated `Cargo.toml` file:

Filename: `Cargo.toml`

```
[package]  
name = "guessing_game"  
version = "0.1.0"  
authors = ["Your Name <you@example.com>"]  
  
[dependencies]
```

If the author information that Cargo obtained from your environment is not what you want, edit the file and save it again.

As you saw in Chapter 1, `cargo new` generates a "Hello, world!" program for `src/main.rs` file:

Filename: `src/main.rs`

```
fn main() {  
    println!("Hello, world!");  
}
```

Now let's compile this "Hello, world!" program and run it in the same step using the terminal:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
     Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you need to rapidly iterate on a program, quickly testing each iteration before moving on to the next one.

Reopen the `src/main.rs` file. You'll be writing all the code in this file.

Processing a Guess

The first part of the guessing game program will ask for user input, process it, and then check that the input is in the expected form. To start, we'll allow the player to input their guess in Listing 2-1 into `src/main.rs`.

Filename: `src/main.rs`

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-1: Code that gets a guess from the user and prints it

This code contains a lot of information, so let's go over it line by line. To obtain the user's input and print the result as output, we need to bring the `io` (input/output) library into scope. The `io` library is part of the standard library (which is known as `std`):

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

By default, Rust brings only a few types into the scope of every program in `std`. If you want to use a type that isn't in the prelude, you have to bring that type into scope explicitly with a `use` statement. Using the `std::io` library provides you with a number of useful functions for reading and writing data to files and accepting user input.

As you saw in Chapter 1, the `main` function is the entry point into the program:

```
fn main() {
```

The `fn` syntax declares a new function, the parentheses, `()`, indicate where the function begins, and the curly bracket, `{`, starts the body of the function.

As you also learned in Chapter 1, `println!` is a macro that prints a string to the console:

```
    println!("Guess the number!");
    println!("Please input your guess.");
```

This code is printing a prompt stating what the game is and requesting input

Storing Values with Variables

Next, we'll create a place to store the user input, like this:

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

Now the program is getting interesting! There's a lot going on in this little line. The `let` statement, which is used to create a *variable*. Here's another example:

```
let foo = bar;
```

This line creates a new variable named `foo` and binds it to the value of the `bar` variable. Variables are immutable by default. We'll be discussing this concept in detail in the "Mutability" section in Chapter 3. The following example shows how to use `mut` to make a variable mutable:

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

Note: The `//` syntax starts a comment that continues until the end of the line. Comments are a great way to add notes to your code, explaining what everything in comments, which are discussed in more detail in Chapter 3.

Now you know that `let mut guess` will introduce a mutable variable named `guess`. The value on the right side of the equal sign (`=`) is the value that `guess` is bound to, which is the result of calling the `String::new`, a function that returns a new instance of a `String`. `String` is a type defined by the standard library that is a growable, UTF-8 encoded bit of text.

The `::` syntax in the `::new` line indicates that `new` is an *associated function*. Associated functions are implemented on a type, in this case `String`, rather than on the type itself. Some languages call this a *static method*.

This `new` function creates a new, empty string. You'll find a `new` function on many types, such as `String`, because it's a common name for a function that makes a new value of some kind.

To summarize, the `let mut guess = String::new();` line has created a mutable variable named `guess` that is currently bound to a new, empty instance of a `String`. Whew!

Recall that we included the input/output functionality from the standard library on the first line of the program. Now we'll call an associated function, `stdin`, to read input from the terminal.

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

If we hadn't listed the `use std::io` line at the beginning of the program, we would have had to use the full path to the `read_line` function, `std::io::stdin::read_line`. The `stdin` function returns an instance of the `Read` trait, which is a type that represents a handle to the standard input for your terminal.

The next part of the code, `.read_line(&mut guess)`, calls the `read_line` method on the `Read` input handle to get input from the user. We're also passing one argument to the `read_line` method: `&mut guess`.

The job of `read_line` is to take whatever the user types into standard input and return it as a `String`. Since the user types a string, the `read_line` method takes a `String` as an argument. The `String` argument needs to be mutable, so we pass `&mut guess`.

method can change the string's content by adding the user input.

The `&` indicates that this argument is a *reference*, which gives you a way to let code access one piece of data without needing to copy that data into memory. References are a complex feature, and one of Rust's major advantages is how it handles them. You don't need to know a lot of those details to finish this problem; what you need to know is that like variables, references are immutable by default. Heres how we can use `&mut` rather than `&` to make it mutable. (Chapter 4 will explain more about references and how to use them thoroughly.)

Handling Potential Failure with the `Result` Type

We're not quite done with this line of code. Although what we've discussed so far is correct, it's only the first part of the single logical line of code. The second part is:

```
.expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, it's often wise to introduce whitespace to help break up long lines. We could have written this code as:

```
io::stdin().read_line(&mut guess).expect("Failed to read line");
```

However, one long line is difficult to read, so it's best to divide it: two lines follow. Let's discuss what this line does.

As mentioned earlier, `read_line` puts what the user types into the string we receive. It also returns a value—in this case, an `io::Result`. Rust has a number of types named after error handling, such as `Result` and `Option`, which are part of the standard library: a generic `Result` as well as specific versions for submodules.

The `Result` types are *enumerations*, often referred to as *enums*. An enumeration is a type that represents a set of values. In this case, the `Result` type has two variants: `Ok` and `Err`. These variants have a fixed set of values, and those values are called the enum's *variants*. Chapter 4 will go into more detail.

For `Result`, the variants are `Ok` or `Err`. The `Ok` variant indicates the operation was successful, and the value inside `Ok` is the successfully generated value. The `Err` variant means the operation failed and contains information about how or why the operation failed.

The purpose of these `Result` types is to encode error-handling information in a type-safe way. Just like any type, they have methods defined on them. An instance of `io::Result` has a `unwrap` method that you can call. If this instance of `io::Result` is an `Err` value, executing `unwrap` would cause your program to crash and display the message that you passed as an argument. For example, if you call `read_line` and it returns an `Err`, it would likely be the result of an error from the underlying operating system. If this instance of `io::Result` is an `Ok` value, then `unwrap` will return the value that `Ok` is holding and return just that value to you so you can use it. For example, if you call `read_line` and it returns an `Ok` value, the value is the number of bytes in what the user entered into standard input.

If you don't call `expect`, the program will compile, but you'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `std::result::Result` which must be used
  --> src/main.rs:10:5
    |
10 |     io::stdin().read_line(&mut guess);
    |     ^^^^^^^^^^^^^^^^^^
    |
    = note: #[warn(unused_must_use)] on by default
```

Rust warns that you haven't used the `Result` value returned from `read_line`. Your program hasn't handled a possible error.

The right way to suppress the warning is to actually write error handling, but if you crash this program when a problem occurs, you can use `expect`. You'll learn more about errors in Chapter 9.

Printing Values with `println!` Placeholders

Aside from the closing curly brackets, there's only one more line to discuss in this section, which is the following:

```
println!("You guessed: {}", guess);
```

This line prints the string we saved the user's input in. The set of curly brackets `{}` think of `{}` as little crab pincers that hold a value in place. You can print more than one value separated by commas. You can also have multiple sets of curly brackets: the first set of curly brackets holds the first value listed after the string, the second set holds the second value, and so on. Printing multiple values in one line of code would look like this:

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

This code would print `x = 5 and y = 10`.

Testing the First Part

Let's test the first part of the guessing game. Run it using `cargo run`:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
        Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

At this point, the first part of the game is done: we're getting input from the user and printing it.

Generating a Secret Number

Next, we need to generate a secret number that the user will try to guess. This number needs to be different every time so the game is fun to play more than once. Let's use a random number generator to generate a number between 1 and 100 so the game isn't too difficult. Rust doesn't yet include random number generation functionality in its standard library. However, the Rust team does provide a crate called `rand` that makes it easy to generate random numbers.

Using a Crate to Get More Functionality

Remember that a crate is a package of Rust code. The project we've been building is an executable. The `rand` crate is a *library crate*, which contains code that can be used by other programs.

Cargo's use of external crates is where it really shines. Before we can write code that depends on `rand`, we need to modify the `Cargo.toml` file to include the `rand` crate as a dependency. Open `Cargo.toml` and add the following line to the bottom beneath the `[dependencies]` section:

Filename: `Cargo.toml`

```
[dependencies]
rand = "0.3.14"
```

In the `Cargo.toml` file, everything that follows a header is part of a section that header defines. Once the header is gone, another section starts. The `[dependencies]` section is where you tell Cargo what other crates your project depends on and which versions of those crates you require. In the `Cargo.toml` file above, we're telling Cargo that our project depends on the `rand` crate with the semantic version specifier `0.3.14`. Cargo understands this as a range of versions (sometimes called *SemVer*), which is a standard for writing version numbers. The range `0.3.14` actually shorthand for `^0.3.14`, which means “any version that has a public release with semantic version `0.3.14`.”

Now, without changing any of the code, let's build the project, as shown in Listing 2-2.

```
$ cargo build
   Updating registry `https://github.com/rust-lang/crates.io-index'
Downloaded rand v0.3.14
Downloaded libc v0.2.14
  Compiling libc v0.2.14
  Compiling rand v0.3.14
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Listing 2-2: The output from running `cargo build` after adding the `rand` crate to `Cargo.toml`.

You may see different version numbers (but they will all be compatible with SemVer!), and the lines may be in a different order.

Now that we have an external dependency, Cargo fetches the latest versions from the `Crates.io` registry, which is a copy of data from [Crates.io](#). Crates.io is where people in the Rust community publish their open source Rust projects for others to use.

After updating the registry, Cargo checks the `[dependencies]` section and downloads the dependencies it needs. In this case, although we only listed `rand` as a dependency, Cargo also downloaded `libc`, because `rand` depends on `libc` to work. After downloading the dependencies, Cargo compiles them and then compiles the project with the dependencies available.

If you immediately run `cargo build` again without making any changes, you'll see the same output as in Listing 2-2, except you'll see it aside from the `Finished` line. Cargo knows it has already downloaded and compiled the dependencies, and you haven't changed anything about them in your `Cargo.toml` file. Cargo also knows that you haven't changed anything about your code, so it doesn't recommend anything to do, it simply exits.

If you open up the `src/main.rs` file, make a trivial change, and then save it and run `cargo build` again, you'll see two lines of output:

```
$ cargo build
  Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

These lines show Cargo only updates the build with your tiny change to the dependencies: dependencies haven't changed, so Cargo knows it can reuse what it has already compiled for those. It just rebuilds your part of the code.

Ensuring Reproducible Builds with the *Cargo.lock* File

Cargo has a mechanism that ensures you can rebuild the same artifact every time else builds your code: Cargo will use only the versions of the dependencies you indicate otherwise. For example, what happens if next week version `v0.3.14` comes out and contains an important bug fix but also contains a regression code?

The answer to this problem is the *Cargo.lock* file, which was created the first time you ran `cargo build` and is now in your `guessing_game` directory. When you build a project again, Cargo figures out all the versions of the dependencies that fit the criteria and writes them to the *Cargo.lock* file. When you build your project in the future, Cargo will see that the *Cargo.lock* file exists and use the versions specified there rather than doing all the work of finding new versions again. This lets you have a reproducible build automatically. In other words, you can build at `v0.3.14` until you explicitly upgrade, thanks to the *Cargo.lock* file.

Updating a Crate to Get a New Version

When you *do* want to update a crate, Cargo provides another command, `cargo update`. It reads the *Cargo.lock* file and figure out all the latest versions that fit your specifications. If you run `cargo update`, Cargo will write those versions to the *Cargo.lock* file.

But by default, Cargo will only look for versions larger than `v0.3.0` and smaller than `v0.4.0`. If the `rand` crate has released two new versions, `v0.3.15` and `v0.4.0`, you would see the following output if you ran `cargo update`:

```
$ cargo update
Updating registry `https://github.com/rust-lang/crates.io-index'
Updating rand v0.3.14 -> v0.3.15
```

At this point, you would also notice a change in your *Cargo.lock* file noting that the `rand` crate you are now using is `v0.3.15`.

If you wanted to use `rand` version `v0.4.0` or any version in the `v0.4.x` series, you would need to edit the *Cargo.toml* file to look like this instead:

```
[dependencies]
rand = "0.4.0"
```

The next time you run `cargo build`, Cargo will update the registry of crates and reevaluate your `rand` requirements according to the new version you have specified.

There's a lot more to say about Cargo and its ecosystem which we'll discuss in the next chapter, but that's all you need to know. Cargo makes it very easy to reuse libraries, and to write smaller projects that are assembled from a number of packages.

Generating a Random Number

Now that you've added the `rand` crate to *Cargo.toml*, let's start using `rand` to generate random numbers in our application. Update `src/main.rs`, as shown in Listing 2-3:

Filename: src/main.rs

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-3: Adding code to generate a random number

First, we add a line that lets Rust know we'll be using the `rand` crate as an external crate. This also does the equivalent of calling `use rand`, so now we can call anything in the `rand` crate by placing `rand::` before it.

Next, we add another `use` line: `use rand::Rng`. The `Rng` trait defines methods for generating random numbers. The `gen_range` method for number generators implement, and this trait must be in scope for us to use it. Listing 10 will cover traits in detail.

Also, we're adding two more lines in the middle. The `rand::thread_rng` function creates a particular random number generator that we're going to use: one that is local to the thread of execution and seeded by the operating system. Next, we call the `gen_range` method on the random number generator. This method is defined by the `Rng` trait that we added earlier. The `gen_range` method takes two numbers as arguments and generates a random number between them. It's inclusive on the lower bound and exclusive on the upper bound, so we need to specify `1` and `101` to request a number between 1 and 100.

Note: You won't just know which traits to use and which methods and functions to call. Instructions for using a crate are in each crate's documentation. An easy way to find documentation for a crate is to run the `cargo doc --open` command, which will build the documentation for all of the crates in your project and open it in your browser. You can also explore the other functionality in the `rand` crate, for example, run `cargo doc --open` and then click on the `rand` link in the sidebar on the left.

The second line that we added to the code prints the secret number. This is useful for developing the program to be able to test it, but we'll delete it from the final version of the program if the program prints the answer as soon as it starts!

Try running the program a few times:

```
$ cargo run
    Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
        Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

You should get different random numbers, and they should all be numbers! Great job!

Comparing the Guess to the Secret Number

Now that we have user input and a random number, we can compare them. Listing 2-4. Note that this code won't compile quite yet, as we will explain.

Filename: src/main.rs

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    // ---snip---

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

Listing 2-4: Handling the possible return values of comparing two numbers

The first new bit here is another `use` statement, bringing a type called `std::cmp::Ordering` from the standard library. Like `Result`, `Ordering` is another enum, but it has three variants: `Less`, `Greater`, and `Equal`. These are the three outcomes that you compare two values.

Then we add five new lines at the bottom that use the `Ordering` type.

The `cmp` method compares two values and can be called on anything that has a reference to whatever you want to compare with: here it's comparing the `secret_number`. Then it returns a variant of the `Ordering` enum we brought in via the `use` statement. We use a `match` expression to decide what to do next based on which variant of `Ordering` was returned from the call to `cmp` with the values in `guess` and `secret_number`.

A `match` expression is made up of *arms*. An arm consists of a *pattern* and th

run if the value given to the beginning of the `match` expression fits that arm value given to `match` and looks through each arm's pattern in turn. The `match` patterns are powerful features in Rust that let you express a variety of situations and make sure that you handle them all. These features will be covered in Chapter 6 and Chapter 18, respectively.

Let's walk through an example of what would happen with the `match` expression. If the user has guessed 50 and the randomly generated secret number this function compares 50 to 38, the `cmp` method will return `Ordering::Greater`, because the `match` expression gets the `Ordering::Greater` value and starts checking the first arm's pattern, `Ordering::Less`, and sees that the value does not match `Ordering::Less`, so it ignores the code in that arm and moves to the next arm's pattern, `Ordering::Greater`, which does match `Ordering::Greater`! The `as!` arm will execute and print `Too big!` to the screen. The `match` expression needs to look at the last arm in this scenario.

However, the code in Listing 2-4 won't compile yet. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
--> src/main.rs:23:21
  |
23 |     match guess.cmp(&secret_number) {
  |           ^^^^^^^^^^^^^^^^^ expected struct `std::strin
integral variable
  |
  = note: expected type `&std::string::String`
  = note:     found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

The core of the error states that there are *mismatched types*. Rust has a strong type system. However, it also has type inference. When we wrote `let mut guess = String::new()`, Rust was able to infer that `guess` should be a `String` and didn't make us write the type. On the other hand, `secret_number`, is a number type. A few number types can have a value between 0 and 100: `u8`, a 8-bit unsigned integer; `u16`, a 16-bit unsigned integer; `u32`, a 32-bit unsigned integer; `u64`, a 64-bit unsigned integer; `i8`, a 8-bit signed integer; `i16`, a 16-bit signed integer; `i32`, a 32-bit signed integer; `i64`, a 64-bit signed integer; and `f32`, a 32-bit floating-point number. `secret_number` defaults to an `i32`, which is the type of `secret_number` unless you add type annotations. This means that Rust cannot compare a `String` and a number type. The reason for the error is that Rust cannot compare a string and a number type.

Ultimately, we want to convert the `String` the program reads as input into a numerical type so we can compare it numerically to the `secret_number`. We can do that by adding the following code to the `main` function body:

Filename: `src/main.rs`

```
// --snip--  
  
let mut guess = String::new();  
  
io::stdin().read_line(&mut guess)  
    .expect("Failed to read line");  
  
let guess: u32 = guess.trim().parse()  
    .expect("Please type a number!");  
  
println!("You guessed: {}", guess);  
  
match guess.cmp(&secret_number) {  
    Ordering::Less => println!("Too small!"),  
    Ordering::Greater => println!("Too big!"),  
    Ordering::Equal => println!("You win!"),  
}  
}
```

The two new lines are:

```
let guess: u32 = guess.trim().parse()  
    .expect("Please type a number!");
```

We create a variable named `guess`. But wait, doesn't the program already have `guess`? It does, but Rust allows us to *shadow* the previous value of `guess`. This feature is often used in situations in which you want to convert a value from one type to another. Shadowing lets us reuse the `guess` variable name rather than forcing us to use different names for variables, like `guess_str` and `guess` for example. (Chapter 3 covers shadowing in more detail.)

We bind `guess` to the expression `guess.trim().parse()`. The `guess` in the original `guess` was a `String` with the input in it. The `trim` method on `String` eliminates any whitespace at the beginning and end. Although `u32` can contain characters, the user must press enter to satisfy `read_line`. When the user presses enter, a carriage return character is added to the string. For example, if the user types 5 and presses enter, the input is this: `5\n`. The `\n` represents "newline," the result of pressing enter. The `trim` method removes the `\n`, resulting in just `5`.

The `parse` method on `String` parses a string into some kind of number. Because we want to parse a variety of number types, we need to tell Rust the exact number type. We do this by adding a colon (`:`) after `guess`. The colon tells Rust we'll annotate the variable with the type `u32`. The colon (`:`) after `guess` tells Rust we'll annotate the variable with the type `u32`. The `u32` seen here is an unsigned, 32-bit integer. This is a common choice for a small positive number. You'll learn about other number types in Chapter 3. The `u32` annotation in this example program and the comparison with `secret_number` tell Rust that `secret_number` should be a `u32` as well. So now the comparison is comparing two values of the same type!

The call to `parse` could easily cause an error. If, for example, the string contains a character that has no way to convert that to a number. Because it might fail, the `parse` method returns a `Result` type, much as the `read_line` method does (discussed earlier in "Handling the Result Type"). We'll treat this `Result` the same way by using the `expect` method. This method returns an `Err` variant because it couldn't create a number from the input. If `parse` can successfully convert the string to a number, it will return the `Ok` variant of `Result`, and `expect` will return the value from the `Ok` value.

Let's run the program now!

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
76
You guessed: 76
Too big!
```

Nice! Even though spaces were added before the guess, the program still figured out the user guessed 76. Run the program a few times to verify the different behavior with different inputs: guess the number correctly, guess a number that is too high, and guess a number that is too low.

We have most of the game working now, but the user can make only one guess. Let's add a loop!

Allowing Multiple Guesses with Looping

The `loop` keyword creates an infinite loop. We'll add that now to give users the ability to guess the number:

Filename: src/main.rs

```
// --snip--

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

As you can see, we've moved everything into a loop from the `guess` input part. Indent the lines inside the loop another four spaces each and run the program again. There is a new problem because the program is doing exactly what we told it to do: it will guess forever! It doesn't seem like the user can quit!

The user could always halt the program by using the keyboard shortcut `ctrl-C`. Another way to escape this insatiable monster, as mentioned in the `parse` discussion, is to add a `break` statement to the `loop` condition. In the “Guess to the Secret Number”: if the user enters a non-number answer, the program will break out of the loop. The user can take advantage of that in order to quit, as shown here:

```
$ cargo run
Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { I
}', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exi
```

Typing `quit` actually quits the game, but so will any other non-number input. It's suboptimal to say the least. We want the game to automatically stop when the user guesses correctly.

Quitting After a Correct Guess

Let's program the game to quit when the user wins by adding a `break` statement.

Filename: `src/main.rs`

```
// --snip--

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => {
        println!("You win!");
        break;
    }
}
```

Adding the `break` line after `You win!` makes the program exit the loop when the user guesses the secret number correctly. Exiting the loop also means exiting the program, because it's the last part of `main`.

Handling Invalid Input

To further refine the game's behavior, rather than crashing the program when the user enters an invalid number, let's make the game ignore a non-number so the user can continue playing. We can do this by altering the line where `guess` is converted from a `String` to a `u32`:

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

Switching from an `expect` call to a `match` expression is how you generally map an error to handling the error. Remember that `parse` returns a `Result` type enum that has the variants `Ok` or `Err`. We're using a `match` expression here to handle the ordering result of the `cmp` method.

If `parse` is able to successfully turn the string into a number, it will return an `Ok` value containing the resulting number. That `Ok` value will match the first arm's pattern, and the `num` variable will be bound to the value produced by `parse`. If `parse` fails, it will just return the `Err` value that `parse` produced and put inside the `Err` variant of the `Result` enum. This means we can just ignore the `Err` value and move on to the next iteration of the loop.

If `parse` is *not* able to turn the string into a number, it will return an `Err` value containing information about the error. The `Err` value does not match the `Ok(num)` pattern in the first arm, but it does match the `Err(_)` pattern in the second arm. The underscore `_` matches any value; in this example, we're saying we want to match all `Err` values, no matter what kind of error they have inside them. So the program will execute the second arm's code, go to the next iteration of the `loop`, and ask for another guess. So effectively, we're catching all errors that `parse` might encounter!

Now everything in the program should work as expected. Let's try it:

```
$ cargo run
     Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny final tweak, we will finish the guessing game. Recall that we were printing the secret number. That worked well for testing, but it ruins the game. Instead, we'll use `println!` that outputs the secret number. Listing 2-5 shows the final code:

Filename: `src/main.rs`

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Listing 2-5: Complete guessing game code

Summary

At this point, you've successfully built the guessing game! Congratulations!

This project was a hands-on way to introduce you to many new Rust concepts, methods, associated functions, the use of external crates, and more. In the rest of the book, you'll learn about these concepts in more detail. Chapter 3 covers concepts that most languages have, such as variables, data types, and functions, and shows how they're used in Rust. Chapter 4 explores ownership, a feature that makes Rust different from other languages. Chapter 5 discusses structs and method syntax, and Chapter 6 explains how enums work.

Common Programming Concepts

This chapter covers concepts that appear in almost every programming language. Many programming languages have much in common at their core. Some of the concepts presented in this chapter are unique to Rust, but we'll discuss them in the context of how they relate to other languages. We'll explain the conventions around using these concepts.

Specifically, you'll learn about variables, basic types, functions, comments, and more. These concepts are foundations will be in every Rust program, and learning them early will give you a solid starting point to build upon.

Keywords

The Rust language has a set of *keywords* that are reserved for use by the language, just as in other languages. Keep in mind that you cannot use these words as regular identifiers or functions. Most of the keywords have special meanings, and you'll be using them frequently in your Rust programs; a few have no current functionality associated with them. Some keywords have been reserved for functionality that might be added to Rust in the future. You can find a list of the keywords in Appendix A.

Variables and Mutability

As mentioned in Chapter 2, by default variables are immutable. This is one of the safety features of Rust that gives you the ability to write your code in a way that takes advantage of the safety and efficiency guarantees that Rust offers. However, you still have the option to make your variables mutable if you need to, and why Rust encourages you to favor immutability and why sometimes you might want to use mutable variables.

When a variable is immutable, once a value is bound to a name, you can't change it. To illustrate this, let's generate a new project called *variables* in your *projects* directory and add some code to it:

Then, in your new *variables* directory, open *src/main.rs* and replace its code with the following code. It won't compile just yet:

Filename: *src/main.rs*

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Save and run the program using `cargo run`. You should receive an error message similar to the following:

```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
 |
2 |     let x = 5;
3 |     - first assignment to `x`
4 |     println!("The value of x is: {}", x);
|     ^^^^^ cannot assign twice to immutable variable
```

This example shows how the compiler helps you find errors in your program. While these errors can be frustrating, they only mean your program isn't safely doing what you intended. They do not mean that you're not a good programmer! Experienced Rustaceans know that these errors are common.

The error indicates that the cause of the error is that you tried to assign a value to the immutable variable `x`, because you tried to assign a value to a variable that was previously assigned a value.

It's important that we get compile-time errors when we attempt to change a variable that was designated as immutable because this very situation can lead to bugs. If one

operates on the assumption that a value will never change and another part that value, it's possible that the first part of the code won't do what it was de of this kind of bug can be difficult to track down after the fact, especially wh code changes the value only *sometimes*.

In Rust, the compiler guarantees that when you state that a value won't char change. That means that when you're reading and writing code, you don't ha and where a value might change. Your code is thus easier to reason through

But mutability can be very useful. Variables are immutable only by default; a you can make them mutable by adding `mut` in front of the variable name. If this value to change, `mut` conveys intent to future readers of the code by inc of the code will be changing this variable value.

For example, let's change `src/main.rs` to the following:

Filename: `src/main.rs`

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

When we run the program now, we get this:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

We're allowed to change the value that `x` binds to from 5 to 6 when `mut` i you'll want to make a variable mutable because it makes the code more con had only immutable variables.

There are multiple trade-offs to consider in addition to the prevention of bu where you're using large data structures, mutating an instance in place may and returning newly allocated instances. With smaller data structures, creati writing in a more functional programming style may be easier to think throu performance might be a worthwhile penalty for gaining that clarity.

Differences Between Variables and Constants

Being unable to change the value of a variable might have reminded you of a concept that most other languages have: *constants*. Like immutable variable that are bound to a name and are not allowed to change, but there are a few constants and variables.

First, you aren't allowed to use `mut` with constants. Constants aren't just im—they're always immutable.

You declare constants using the `const` keyword instead of the `let` keyword. value *must* be annotated. We're about to cover types and type annotations in "Types," so don't worry about the details right now. Just know that you must a type.

Constants can be declared in any scope, including the global scope, which means values that many parts of code need to know about.

The last difference is that constants may be set only to a constant expression or function call or any other value that could only be computed at runtime.

Here's an example of a constant declaration where the constant's name is MAX_POINTS and is set to 100,000. (Rust's constant naming convention is to use all uppercase between words):

```
const MAX_POINTS: u32 = 100_000;
```

Constants are valid for the entire time a program runs, within the scope they're declared in, making them a useful choice for values in your application domain that multiple parts of your program might need to know about, such as the maximum number of points allowed to earn or the speed of light.

Naming hardcoded values used throughout your program as constants is useful because it gives meaning of that value to future maintainers of the code. It also helps to have a single place to change if the hardcoded value needed to be updated.

Shadowing

As you saw in the "Comparing the Guess to the Secret Number" section in Chapter 4, shadowing is when you declare a new variable with the same name as a previous variable, and the new variable shadows the previous variable. Rustaceans say that the first variable is *shadowed* by the second. This means that the second variable's value is what appears when the variable is used. We can demonstrate shadowing by using the same variable's name and repeating the use of the `let` keyword as follows:

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
```

This program first binds `x` to a value of `5`. Then it shadows `x` by repeating the declaration. The second `let` adds `1` to the original value, so the value of `x` is then `6`. The third `let` statement multiplies the previous value by `2` to give `x` a final value of `12`. When we run the program, we output the following:

```
$ cargo run
Compiling variables v0.1.0 (file:///projects/variables)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/variables`
The value of x is: 12
```

Shadowing is different than marking a variable as `mut`, because we'll get a compile error if we accidentally try to reassign to this variable without using the `let` keyword. It's similar to mutation in that we can perform a few transformations on a value but have the variable be immutable until those transformations have been completed.

The other difference between `mut` and shadowing is that because we're effectively creating a new variable, we can do things like this:

variable when we use the `let` keyword again, we can change the type of the same name. For example, say our program asks a user to show how many spaces between some text by inputting space characters, but we really want to store a number:

```
let spaces = "    ";
let spaces = spaces.len();
```

This construct is allowed because the first `spaces` variable is a string type and the second is a number type. Shadowing thus spares us from having to come up with different names like `spaces_str` and `spaces_num`; instead, we can reuse the simpler `spaces` name to use `mut` for this, as shown here, we'll get a compile-time error:

```
let mut spaces = "    ";
spaces = spaces.len();
```

The error says we're not allowed to mutate a variable's type:

```
error[E0308]: mismatched types
--> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |     ^^^^^^^^^^^^^ expected &str, found usize
  |
  = note: expected type `&str`
        found type `usize`
```

Now that we've explored how variables work, let's look at more data types that we can use.

Data Types

Every value in Rust is of a certain *data type*, which tells Rust what kind of data it is and how to work with that data. We'll look at two data type subsets: scalar and aggregate.

Keep in mind that Rust is a *statically typed* language, which means that it must know exactly what type each variable is at compile time. The compiler can usually infer what type we want from context, such as when we convert a `String` to a numeric type using `parse` in the “Comparing the Guess to the User Input” section. However, in cases when many types are possible, such as when we convert a `String` to a numeric type using `parse` in the “Comparing the Guess to the User Input” section, we must add a type annotation, like this:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

If we don't add the type annotation here, Rust will display the following error:

```
error[E0282]: type annotations needed
--> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |     ^
  |     |
  |     cannot infer type for `guess`
  |     consider giving `guess` a type
```

You'll see different type annotations for other data types.

Scalar Types

A *scalar* type represents a single value. Rust has four primary scalar types: integers, Booleans, and characters. You may recognize these from other programming languages. Let's jump into how they work in Rust.

Integer Types

An *integer* is a number without a fractional component. We used one integer type, `u32`. This type declaration indicates that the value it's associated with is an unsigned integer (signed integer types start with `i`, instead of `u`) that takes up 32 bits. The following table shows the built-in integer types in Rust. Each variant in the Signed and Unsigned columns (for example, `i16`) can be used to declare the type of an integer value.

Table 3-1: Integer Types in Rust

Length	Signed	Unsigned
8-bit	<code>i8</code>	<code>u8</code>
16-bit	<code>i16</code>	<code>u16</code>
32-bit	<code>i32</code>	<code>u32</code>
64-bit	<code>i64</code>	<code>u64</code>
128-bit	<code>i128</code>	<code>u128</code>
arch	<code>isize</code>	<code>usize</code>

Each variant can be either signed or unsigned and has an explicit size. *Signed* means whether it's possible for the number to be negative or positive—in other words, does the number need a sign with it (signed) or whether it will only ever be positive (unsigned). Therefore, signed variants can therefore be represented without a sign (unsigned). It's like writing numbers: in mathematics, a number is shown with a plus sign or a minus sign; however, in computer memory, the number is positive, it's shown with no sign. Signed numbers are stored using two's complement representation (if you're unsure what this is, you can search for it online; an explanation is beyond the scope of this book).

Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive, where n is the number of bits that variant uses. So an `i8` can store numbers from $-(2^7)$ to $2^7 - 1$, which is from -128 to 127. Unsigned variants can store numbers from 0 to $2^n - 1$, so a `u8` can store numbers from 0 to 255, which equals 0 to 255.

Additionally, the `isize` and `usize` types depend on the kind of computer you're using: 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.

You can write integer literals in any of the forms shown in Table 3-2. Note that the suffixes `u` and `l` except the byte literal allow a type suffix, such as `57u8`, and `_` as a visual separator.

Table 3-2: Integer Literals in Rust

Number literals	Example
Decimal	<code>98_222</code>
Hex	<code>0xff</code>
Octal	<code>0o77</code>
Binary	<code>0b1111_0000</code>
Byte (<code>u8</code> only)	<code>b'A'</code>

So how do you know which type of integer to use? If you're unsure, Rust's de good choices, and integer types default to `i32`: this type is generally the fas systems. The primary situation in which you'd use `isize` or `usize` is when collection.

Integer Overflow

Let's say that you have a `u8`, which can hold values between zero and `255`. to change it to `256`? This is called "integer overflow", and Rust has some inter this behavior. When compiling in debug mode, Rust checks for this kind of is program to *panic*, which is the term Rust uses when a program exits with an panics more in Chapter 9.

In release builds, Rust does not check for overflow, and instead will do some compliment wrapping." In short, `256` becomes `0`, `257` becomes `1`, etc. Rel considered an error, even if this behavior happens. If you want this behavior library has a type, `Wrapping`, that provides it explicitly.

Floating-Point Types

Rust also has two primitive types for *floating-point numbers*, which are numb Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in default type is `f64` because on modern CPUs it's roughly the same speed as more precision.

Here's an example that shows floating-point numbers in action:

Filename: src/main.rs

```
fn main() {
    let x = 2.0; // f64
    let y: f32 = 3.0; // f32
}
```

Floating-point numbers are represented according to the IEEE-754 standard. single-precision float, and `f64` has double precision.

Numeric Operations

Rust supports the basic mathematical operations you'd expect for all of the i subtraction, multiplication, division, and remainder. The following code shov one in a `let` statement:

Filename: src/main.rs

```

fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;

    // remainder
    let remainder = 43 % 5;
}

```

Each expression in these statements uses a mathematical operator and evaluates which is then bound to a variable. Appendix B contains a list of all operators

The Boolean Type

As in most other programming languages, a Boolean type in Rust has two possible values: `true` and `false`. The Boolean type in Rust is specified using `bool`. For example:

Filename: src/main.rs

```

fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}

```

The main way to consume Boolean values is through conditionals, such as a `if` expression. We'll cover how `if` expressions work in Rust in the "Control Flow" section.

Booleans are one byte in size.

The Character Type

So far we've worked only with numbers, but Rust supports letters too. Rust's character type is called `char`, and the following code shows one way to use it. Note that the `char` literal is specified with single quotes, as opposed to string literals, which are specified with double quotes.)

Filename: src/main.rs

```

fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}

```

Rust's `char` type represents a Unicode Scalar Value, which means it can represent more than just ASCII. Accented letters; Chinese, Japanese, and Korean characters; emojis; and other special characters are all valid `char` values in Rust. Unicode Scalar Values range from `U+0000` to `U+10FFFF` inclusive. However, a "character" isn't really a concept in Unicode; its intuition for what a "character" is may not match up with what a `char` is in Rust. This topic is covered in detail in "Strings" in Chapter 8.

Compound Types

Compound types can group multiple values into one type. Rust has two primitive types and arrays.

The Tuple Type

A tuple is a general way of grouping together some number of other values into one compound type. Tuples have a fixed length: once declared, they cannot change size.

We create a tuple by writing a comma-separated list of values inside parentheses. The tuple has a type, and the types of the different values in the tuple don't have to be the same. We've added optional type annotations in this example:

Filename: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

The variable `tup` binds to the entire tuple, because a tuple is considered a single element. To get the individual values out of a tuple, we can use pattern matching on the tuple value, like this:

Filename: src/main.rs

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

This program first creates a tuple and binds it to the variable `tup`. It then uses pattern matching to take `tup` and turn it into three separate variables, `x`, `y`, and `z`. This is called `destructuring`, because it breaks the single tuple into three parts. Finally, the program prints the value of `y`.

In addition to destructuring through pattern matching, we can access a tuple's elements using a period (`.`) followed by the index of the value we want to access. For example:

Filename: src/main.rs

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
}
```

This program creates a tuple, `x`, and then makes new variables for each element at its index. As with most programming languages, the first index in a tuple is 0.

The Array Type

Another way to have a collection of multiple values is with an *array*. Unlike a tuple,

an array must have the same type. Arrays in Rust are different from arrays in C because arrays in Rust have a fixed length, like tuples.

In Rust, the values going into an array are written as a comma-separated list

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Arrays are useful when you want your data allocated on the stack rather than the heap (we'll discuss the stack and the heap more in Chapter 4), or when you want to ensure a fixed number of elements. An array isn't as flexible as the vector type, though. As a collection type provided by the standard library that is allowed to grow or shrink, if you're unsure whether to use an array or a vector, you should probably use a vector and vectors in more detail.

An example of when you might want to use an array rather than a vector is if you know the names of the months of the year. It's very unlikely that such a program will ever need to add or remove months, so you can use an array because you know it will always have the same size.

```
let months = ["January", "February", "March", "April", "May", "June",
              "July", "August", "September", "October", "November", "December"];
```

Arrays have an interesting type; it looks like this: `[type; number]`. For example:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

First, there's square brackets; they look like the syntax for creating an array. Then there's a colon and a semicolon, separating the type information from the length information. The first is the type of each element in the array. Since all elements have the same type, we only need to list it once. After the colon is the type of each element, and after the semicolon is the number that indicates the length of the array. Since an array has a fixed size, it cannot grow or shrink.

Accessing Array Elements

An array is a single chunk of memory allocated on the stack. You can access individual elements using indexing, like this:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

In this example, the variable named `first` will get the value `1`, because that's the value at index `0` in the array. The variable named `second` will get the value `2` from index `1`.

Invalid Array Element Access

What happens if you try to access an element of an array that is past the end of the array? To see what happens, change the example to the following code, which will compile but exit with a non-zero status code:

Filename: src/main.rs

```

fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    println!("The value of element is: {}", element);
}

```

Running this code using `cargo run` produces the following result:

```

$ cargo run
Compiling arrays v0.1.0 (file:///projects/arrays)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/arrays`
thread '' panicked at 'index out of bounds: the len is 5 but
10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.

```

The compilation didn't produce any errors, but the program resulted in a *run* exit successfully. When you attempt to access an element using indexing, Rust index you've specified is less than the array length. If the index is greater than panic.

This is the first example of Rust's safety principles in action. In many low-level check is not done, and when you provide an incorrect index, invalid memory protects you against this kind of error by immediately exiting instead of allowing and continuing. Chapter 9 discusses more of Rust's error handling.

Functions

Functions are pervasive in Rust code. You've already seen one of the most important language: the `main` function, which is the entry point of many programs. You keyword, which allows you to declare new functions.

Rust code uses *snake case* as the conventional style for function and variable names: all letters are lowercase and underscores separate words. Here's a program with an example function definition:

Filename: `src/main.rs`

```

fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}

```

Function definitions in Rust start with `fn` and have a set of parentheses after the function name. The curly brackets tell the compiler where the function body begins and ends.

We can call any function we've defined by entering its name followed by a set of parentheses. Because `another_function` is defined in the program, it can be called from anywhere. Note that we defined `another_function` *after* the `main` function in this program, but we could have defined it before as well. Rust doesn't care where you define your functions as long as they're defined somewhere.

Let's start a new binary project named `functions` to explore functions further. Create another `function` example in `src/main.rs` and run it. You should see the following output:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28 secs
        Running `target/debug/functions`
Hello, world!
Another function.
```

The lines execute in the order in which they appear in the `main` function. First, the `Hello, world!` message prints, and then `another_function` is called and its message is printed.

Function Parameters

Functions can also be defined to have *parameters*, which are special variable declarations within the function's signature. When a function has parameters, you can provide it with concrete values when you call it. Technically, the concrete values are called *arguments*, but people tend to use the words *parameter* and *argument* interchangeably for the variables declared in the function's definition or the concrete values passed in when you call a function.

The following rewritten version of `another_function` shows what parameters it takes:

Filename: `src/main.rs`

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}
```

Try running this program; you should get the following output:

```
$ cargo run
    Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 1.21 secs
        Running `target/debug/functions`
The value of x is: 5
```

The declaration of `another_function` has one parameter named `x`. The type of `x` is `i32`. When `5` is passed to `another_function`, the `println!` macro puts `5` in the position where the braces were in the format string.

In function signatures, you *must* declare the type of each parameter. This is a key part of Rust's design: requiring type annotations in function definitions means the compiler needs you to use them elsewhere in the code to figure out what you mean.

When you want a function to have multiple parameters, separate the parameter names by commas, like this:

Filename: `src/main.rs`

```

fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}

```

This example creates a function with two parameters, both of which are `i32` then prints the values in both of its parameters. Note that function parameters can be the same type, they just happen to be in this example.

Let's try running this code. Replace the program currently in your `functions` project with the preceding example and run it using `cargo run`:

```

$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
Running `target/debug/functions`
The value of x is: 5
The value of y is: 6

```

Because we called the function with `5` as the value for `x` and `6` is passed as the value for `y`, two strings are printed with these values.

Function Bodies

Function bodies are made up of a series of statements optionally ending in a semicolon. So far we've only covered functions without an ending expression, but you have seen examples of statements. Because Rust is an expression-based language, this is an important distinction to understand. Other languages don't have the same distinctions, so let's look at how statements and expressions are similar and how their differences affect the bodies of functions.

Statements and Expressions

We've actually already used statements and expressions. *Statements* are instructions that perform some action and do not return a value. *Expressions* evaluate to a resulting value. Let's look at some examples.

Creating a variable and assigning a value to it with the `let` keyword is a statement:

Filename: `src/main.rs`

```

fn main() {
    let y = 6;
}

```

Listing 3-1: A `main` function declaration containing one statement

Function definitions are also statements; the entire preceding example is a single statement.

Statements do not return values. Therefore, you can't assign a `let` statement a value, as the following code tries to do; you'll get an error:

Filename: `src/main.rs`

```
fn main() {
    let x = (let y = 6);
}
```

When you run this program, the error you'll get looks like this:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found statement (`let`)
--> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |           ^^^
  |
= note: variable declaration using `let` is a statement
```

The `let y = 6` statement does not return a value, so there isn't anything to different from what happens in other languages, such as C and Ruby, where the value of the assignment. In those languages, you can write `x = y = 6` a have the value `6`; that is not the case in Rust.

Expressions evaluate to something and make up most of the rest of the code. Consider a simple math operation, such as `5 + 6`, which is an expression that evaluates to `11`. Expressions can be part of statements: in Listing 3-1, the `6` in the statement `let x = 6` is an expression that evaluates to the value `6`. Calling a function is an expression that evaluates to something. The block that we use to create new scopes, `{}`, is an expression that evaluates to something.

Filename: src/main.rs

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

This expression:

```
{
    let x = 3;
    x + 1
}
```

is a block that, in this case, evaluates to `4`. That value gets bound to `y` as part of the statement. Note the `x + 1` line without a semicolon at the end, which is unusual for you've seen so far. Expressions do not include ending semicolons. If you add a semicolon after an expression, you turn it into a statement, which will then not return a value. You'll learn more about expressions as you explore function return values and expressions next.

Functions with Return Values

Functions can return values to the code that calls them. We don't name return values, but we can declare their type after an arrow (`->`). In Rust, the return value of the function is the value of the final expression in the block or the body of a function. You can return a value from a function by using the `return` keyword and specifying a value, but most functions return the value of the final expression.

expression implicitly. Here's an example of a function that returns a value:

Filename: src/main.rs

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

There are no function calls, macros, or even `let` statements in the `five` function by itself. That's a perfectly valid function in Rust. Note that the function's return type is also `-> i32`. Try running this code; the output should look like this:

```
$ cargo run
Compiling functions v0.1.0 (file:///projects/functions)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/functions`
The value of x is: 5
```

The `5` in `five` is the function's return value, which is why the return type is `i32`. In more detail. There are two important bits: first, the line `let x = five();` is the return value of a function to initialize a variable. Because the function `five`'s return type is `i32`, this line is the same as the following:

```
let x = 5;
```

Second, the `five` function has no parameters and defines the type of the returned value. The return value of the function is a lonely `5` with no semicolon because it's an expression whose value is returned.

Let's look at another example:

Filename: src/main.rs

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Running this code will print `The value of x is: 6`. But if we place a semicolon after the line containing `x + 1`, changing it from an expression to a statement, we'll get an error:

Filename: src/main.rs

```

fn main() {
    let x = plus_one(5);

    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1;
}

```

Compiling this code produces an error, as follows:

```

error[E0308]: mismatched types
--> src/main.rs:7:28
 |
7 |     fn plus_one(x: i32) -> i32 {
|     |-----^
8 |     |         x + 1;
|     |         - help: consider removing this semicolon
9 |     |
|     |     }           ^ expected i32, found ()
|     |
= note: expected type `i32`
        found type `()`
```

The main error message, “mismatched types,” reveals the core issue with this code. The function `plus_one` says that it will return an `i32`, but statements don’t return anything, which is expressed by `()`, the empty tuple. Therefore, nothing is returned, which causes a mismatch between the function definition and results in an error. In this output, Rust provides a message to help rectify this issue: it suggests removing the semicolon, which would fix the error.

Comments

All programmers strive to make their code easy to understand, but sometimes it’s necessary to add notes that aren’t part of the program logic. In these cases, programmers leave notes, or *comments*, in their source code that the compiler will ignore but people reading the source code may find useful.

Here’s a simple comment:

```
// Hello, world.
```

In Rust, comments must start with two slashes and continue until the end of the line. If you want comments that extend beyond a single line, you’ll need to include `//` on each line, like this:

```
// So we're doing something complicated here, long enough that we
// need multiple lines of comments to do it! Whew! Hopefully, this comment
// explains what's going on.
```

Comments can also be placed at the end of lines containing code:

Filename: `src/main.rs`

```

fn main() {
    let lucky_number = 7; // I'm feeling lucky today.
}
```

But you’ll more often see them used in this format, with the comment on a separate line:

code it's annotating:

Filename: src/main.rs

```
fn main() {  
    // I'm feeling lucky today.  
    let lucky_number = 7;  
}
```

Rust also has another kind of comment, documentation comments, which we'll cover in chapter 14.

Control Flow

Deciding whether or not to run some code depending on if a condition is true or false, and running some code repeatedly while a condition is true are basic building blocks in most programming languages. The most common constructs that let you control the flow of execution are `if` expressions and loops.

if Expressions

An `if` expression allows you to branch your code depending on conditions. and then state, "If this condition is met, run this block of code. If the condition is not met, skip this block of code."

Create a new project called *branches* in your *projects* directory to explore the *src/main.rs* file, input the following:

Filename: src/main.rs

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

All `if` expressions start with the keyword `if`, which is followed by a condition that checks whether or not the variable `number` has a value less than 5. The code we want to execute if the condition is true is placed immediately after the condition, enclosed in brackets. Blocks of code associated with the conditions in `if` expressions are called *arms*, just like the arms in `match` expressions that we discussed in the “Comparing Secret Number” section of Chapter 2.

Optionally, we can also include an `else` expression, which we chose to do here as an alternative block of code to execute should the condition evaluate to false. If the `else` expression and the condition is false, the program will just skip the `if` block and move on to the next bit of code.

Try running this code; you should see the following output:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
    Running `target/debug/branches`
condition was true
```

Let's try changing the value of `number` to a value that makes the condition `f` happens:

```
let number = 7;
```

Run the program again, and look at the output:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
    Running `target/debug/branches`
condition was false
```

It's also worth noting that the condition in this code *must* be a `bool`. If the condition is not a `bool`, we'll get an error. For example:

Filename: src/main.rs

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

The `if` condition evaluates to a value of `3` this time, and Rust throws an error:

```
error[E0308]: mismatched types
--> src/main.rs:4:8
 |
4 |     if number {
|     ^^^^^^ expected bool, found integral variable
|
= note: expected type `bool`
       found type `{integer}`
```

The error indicates that Rust expected a `bool` but got an integer. Unlike languages like C and JavaScript, Rust will not automatically try to convert non-Boolean types to `bool`. To make sure the condition is always a Boolean, we need to be explicit and always provide `if` with a Boolean as its condition. If we want the code to only run when a number is not equal to `0`, for example, we can change the code to the following:

Filename: src/main.rs

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("number was something other than zero");
    }
}
```

Running this code will print `number was something other than zero`.

Handling Multiple Conditions with `else if`

You can have multiple conditions by combining `if` and `else` in an `else if` example:

Filename: `src/main.rs`

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

This program has four possible paths it can take. After running it, you should output:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
    Running `target/debug/branches`
number is divisible by 3
```

When this program executes, it checks each `if` expression in turn and executes which the condition holds true. Note that even though 6 is divisible by 2, we don't see the `number is divisible by 2` message because the `else if` condition fails. We also don't see the `number is not divisible by 4, 3, or 2` message because the `else` block. That's because Rust only executes the block for the first true condition; if none of the conditions are true, it doesn't even check the rest.

Using too many `else if` expressions can clutter your code, so if you have many conditions, you might want to refactor your code. Chapter 6 describes a powerful Rust branching construct called `match` for these cases.

Using `if` in a `let` Statement

Because `if` is an expression, we can use it on the right side of a `let` statement:

Filename: `src/main.rs`

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        6
    };

    println!("The value of number is: {}", number);
}
```

Listing 3-2: Assigning the result of an `if` expression to a variable

The `number` variable will be bound to a value based on the outcome of the `if` expression. Run the code to see what happens:

```
$ cargo run
Compiling branches v0.1.0 (file:///projects/branches)
Finished dev [unoptimized + debuginfo] target(s) in 0.30 secs
Running `target/debug/branches`
The value of number is: 5
```

Remember that blocks of code evaluate to the last expression in them, and they are also expressions. In this case, the value of the whole `if` expression depends on which arm of the `if` statement executes. This means the values that have the potential to be results from both arms must be the same type; in Listing 3-2, the results of both the `if` arm and the `else` arm must be integers. If the types are mismatched, as in the following example, we'll get an error:

Filename: `src/main.rs`

```
fn main() {
    let condition = true;

    let number = if condition {
        5
    } else {
        "six"
    };

    println!("The value of number is: {}", number);
}
```

When we try to compile this code, we'll get an error. The `if` and `else` arms are incompatible, and Rust indicates exactly where to find the problem in the code:

```
error[E0308]: if and else have incompatible types
--> src/main.rs:4:18
  |
4 |     let number = if condition {
  |     ^-----^
5 |     |         5
6 |     |     } else {
7 |     |         "six"
8 |     |     };
  |     |____^ expected integral variable, found &str
  |
  |= note: expected type `<integer>`
          found type `&str`
```

The expression in the `if` block evaluates to an integer, and the expression in the `else` block evaluates to a string. This won't work because variables must have a single type at compile time: what type the `number` variable is, definitively, so it can verify that the type is valid everywhere we use `number`. Rust wouldn't be able to do that if it had to determine the type only determined at runtime; the compiler would be more complex and would have to make guarantees about the code if it had to keep track of multiple hypothetical types.

Repetition with Loops

It's often useful to execute a block of code more than once. For this task, Rust provides three kinds of loops: `loop`, `while`, and `for`. A loop runs through the code inside the loop body to the end and then starts again from the beginning. To experiment with loops, let's make a new project called `loop`:

Rust has three kinds of loops: `loop`, `while`, and `for`. Let's try each one.

Repeating Code with `loop`

The `loop` keyword tells Rust to execute a block of code over and over again explicitly tell it to stop.

As an example, change the `src/main.rs` file in your `loops` directory to look like

Filename: `src/main.rs`

```
fn main() {
    loop {
        println!("again!");
    }
}
```

When we run this program, we'll see `again!` printed over and over continually until we explicitly tell it to stop. Most terminals support a keyboard shortcut, `ctrl-c`, to halt a program manually. Most terminals support a keyboard shortcut, `ctrl-c`, to halt a program manually. Most terminals support a keyboard shortcut, `ctrl-c`, to halt a program manually. Most terminals support a keyboard shortcut, `ctrl-c`, to halt a program manually. Most terminals support a keyboard shortcut, `ctrl-c`, to halt a program manually. Give it a try:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.29 secs
Running `target/debug/loops`
again!
again!
again!
again!
again!
^Cagain!
```

The symbol `^C` represents where you pressed `ctrl-c`. You may or may not see `again!` printed after the `^C`, depending on where the code was in the loop when it was interrupted.

Fortunately, Rust provides another, more reliable way to break out of a loop: the `break` keyword within the loop to tell the program when to stop executing the loop. We did this in the guessing game in the “Quitting After a Correct Guess” section to break out of the loop when the user won the game by guessing the correct number.

Returning from loops

One of the uses of a `loop` is to retry an operation you know can fail, such as reading from a file. If the operation fails, the loop will retry until it has completed its job. However, you might need to pass the result of that operation back to the rest of your program. If you add it to the `break` expression you use to stop the loop, it will be passed back to the rest of your program.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

Conditional Loops with `while`

It's often useful for a program to evaluate a condition within a loop. While the condition is true, the loop runs. When the condition ceases to be true, the program calls `break`, so the loop exits.

loop type could be implemented using a combination of `loop`, `if`, `else`, and `break`. That's what you did in Listing 3-2, but what if you wanted to do that now in a program, if you'd like.

However, this pattern is so common that Rust has a built-in language construct for it: `while`. Listing 3-3 uses `while`: the program loops three times, counting down from 3 to 0. After the loop, it prints another message and exits.

Filename: `src/main.rs`

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number = number - 1;
    }

    println!("LIFTOFF!!!");
}
```

Listing 3-3: Using a `while` loop to run code while a condition holds true

This construct eliminates a lot of nesting that would be necessary if you used `loop`, `if`, `else`, and `break`, and it's clearer. While a condition holds true, the code runs; otherwise, it exits.

Looping Through a Collection with `for`

You could use the `while` construct to loop over the elements of a collection, but there's a better way. For example, let's look at Listing 3-4:

Filename: `src/main.rs`

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index = index + 1;
    }
}
```

Listing 3-4: Looping through each element of a collection using a `while` loop

Here, the code counts up through the elements in the array. It starts at index 0 and continues until it reaches the final index in the array (that is, when `index < 5` is no longer true). The output shows that it will print every element in the array:

```
$ cargo run
Compiling loops v0.1.0 (file:///projects/loops)
Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

All five array values appear in the terminal, as expected. Even though `index` is less than 5 at some point, the loop stops executing before trying to fetch a sixth value from the array.

But this approach is error prone; we could cause the program to panic if the incorrect. It's also slow, because the compiler adds runtime code to perform on every element on every iteration through the loop.

As a more concise alternative, you can use a `for` loop and execute some code on every element. A `for` loop looks like this code in Listing 3-5:

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

Listing 3-5: Looping through each element of a collection using a `for` loop

When we run this code, we'll see the same output as in Listing 3-4. More importantly, increased the safety of the code and eliminated the chance of bugs that might occur if we tried to access elements beyond the end of the array or not going far enough and missing some item.

For example, in the code in Listing 3-4, if you removed an item from the `a` array and update the condition to `while index < 4`, the code would panic. Using the `for` loop, there is no need to remember to change any other code if you changed the number of elements in the array.

The safety and conciseness of `for` loops make them the most commonly used loops in Rust. Even in situations in which you want to run some code a certain number of times, it's better to use a `for` loop than a `while` loop. For example, consider the countdown example that used a `while` loop in Listing 3-3, most Rustaceans would prefer to use a `for` loop instead. The way to do that would be to use a `Range`, which is a type provided by the `std::ops` module that generates all numbers in sequence starting from one number and ending before another.

Here's what the countdown would look like using a `for` loop and another range type that we talked about, `rev`, to reverse the range:

Filename: src/main.rs

```
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
```

This code is a bit nicer, isn't it?

Summary

You made it! That was a sizable chapter: you learned about variables, scalar types, functions, comments, `if` expressions, and loops! If you want to practice what you learned, here are some exercises. Most of these exercises build on what was discussed in this chapter, try building programs to do the following:

- Convert temperatures between Fahrenheit and Celsius.
- Generate the nth Fibonacci number.
- Print the lyrics to the Christmas carol "The Twelve Days of Christmas," taking care to handle the repetition in the song.

When you're ready to move on, we'll talk about a concept in Rust that *doesn't* other programming languages: ownership.

Understanding Ownership

Ownership is Rust's most unique feature, and it enables Rust to make memory without needing a garbage collector. Therefore, it's important to understand it in Rust. In this chapter, we'll talk about ownership as well as several related topics, slices, and how Rust lays data out in memory.

What Is Ownership?

Rust's central feature is *ownership*. Although the feature is straightforward to learn, it has significant implications for the rest of the language.

All programs have to manage the way they use a computer's memory while doing so. Most languages have garbage collection that constantly looks for no longer used memory and frees it automatically; in other languages, the programmer must explicitly allocate and free the memory. Rust takes a third approach: memory is managed through a system of ownership with a compiler that checks for errors at compile time. None of the ownership features slow down your program or prevent it from running.

Because ownership is a new concept for many programmers, it does take some time to learn. The good news is that the more experienced you become with Rust and the ownership system, the more you'll be able to naturally develop code that is safe and efficient.

When you understand ownership, you'll have a solid foundation for understanding what makes Rust unique. In this chapter, you'll learn ownership by working through several examples that focus on a very common data structure: strings.

The Stack and the Heap

In many programming languages, you don't have to think about the stack and heap too often. But in a systems programming language like Rust, whether a value is stored on the stack or heap has more of an effect on how the language behaves and why you have to make certain decisions. Parts of ownership will be described in relation to the stack and heap in this chapter, so here is a brief explanation in preparation.

Both the stack and the heap are parts of memory that is available to your program's runtime, but they are structured in different ways. The stack stores values in a specific order: it grows downwards as you add values and shrinks upwards as you remove them. This is referred to as a stack of plates: when you add more plates, you put them on top of the stack; when you need a plate, you take one off the top. Adding or removing plates from the middle of the stack wouldn't work as well! Adding data is called *pushing onto the stack*, and removing it is called *popping off the stack*.

The stack is fast because of the way it accesses the data: it never has to search for a place to store new data or a place to get data from because that place is always the same. The reason that makes the stack fast is that all data on the stack must take up a known amount of space.

Data with a size unknown at compile time or a size that might change can't be stored on the stack. Instead, it must be stored on the heap instead. The heap is less organized: when you put data on the heap, the operating system finds an empty spot somewhere in memory and returns its address to you.

enough, marks it as being in use, and returns a *pointer*, which is the address of the value. This process is called *allocating on the heap*, sometimes abbreviated as just *allocating*. Putting values onto the stack is not considered allocating. Because the pointer is just a reference, you can store the pointer on the stack, but when you want the actual data, you need to dereference the pointer.

Think of being seated at a restaurant. When you enter, you state the number of people in your group, and the staff finds an empty table that fits everyone and leads you to it. If your group comes late, they can ask where you've been seated to find you.

Accessing data in the heap is slower than accessing data on the stack because you have to follow a pointer to get there. Contemporary processors are faster if they just have to access memory directly. Continuing the analogy, consider a server at a restaurant taking orders. It's most efficient to get all the orders at one table before moving on to the next. Taking an order from table A, then an order from table B, then one from C, and then one from B again would be a much slower process. By the same token, a program is better if it works on data that's close to other data (as it is on the stack) rather than scattered away (as it can be on the heap). Allocating a large amount of space on the heap at once can be slow.

When your code calls a function, the values passed into the function (including pointers to data on the heap) and the function's local variables get pushed onto the stack. When the function is over, those values get popped off the stack.

Keeping track of what parts of code are using what data on the heap, minimizing duplicate data on the heap, and cleaning up unused data on the heap so that there's enough space for new data are all problems that ownership addresses. Once you understand ownership, you'll need to think about the stack and the heap very often, but knowing that rust's ownership system exists can help explain why it works the way it does.

Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we walk through some examples that illustrate them:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Variable Scope

We've walked through an example of a Rust program already in Chapter 2. Now that we know the basic syntax, we won't include all the `fn main() {` code in examples, so if you're reading along, you'll have to put the following examples inside a `main` function manually. As a result, the examples will be a bit more concise, letting us focus on the actual details rather than boilerplate code.

As a first example of ownership, we'll look at the *scope* of some variables. A variable's scope is the part of a program for which an item is valid. Let's say we have a variable that looks like this:

```
let s = "hello";
```

The variable `s` refers to a string literal, where the value of the string is hard-coded into our program. The variable is valid from the point at which it's declared until it goes out of scope.

scope. Listing 4-1 has comments annotating where the variable `s` is valid:

```
{
    // s is not valid here, it's not yet declared
    let s = "hello"; // s is valid from this point forward

    // do stuff with s
} // this scope is now over, and s is no longer valid
```

Listing 4-1: A variable and the scope in which it is valid

In other words, there are two important points in time here:

- When `s` comes *into scope*, it is valid.
- It remains valid until it goes *out of scope*.

At this point, the relationship between scopes and when variables are valid is clear. This is true of most programming languages. Now we'll build on top of this understanding by introducing the string type.

The String Type

To illustrate the rules of ownership, we need a data type that is more complex than primitives. We'll cover the basics of ownership in the “Data Types” section of Chapter 3. The types covered previously are simple values that are pushed onto the stack and popped off the stack when their scope is over, but we want to look at something more complex: strings. In this chapter, we'll focus on how Rust handles ownership of strings on the heap and explore how Rust knows when to clean up that data.

We'll use `String` as the example here and concentrate on the parts of strings that involve ownership. These aspects also apply to other complex data types provided by the standard library and that you create. We'll discuss `String` in more depth in Chapter 8.

We've already seen string literals, where a string value is hardcoded into our code. While they're convenient, but they aren't suitable for every situation in which we may want to use them. One reason is that they're immutable. Another is that not every string value can be stored in memory at compile time. For example, what if we want to take user input and store it? For that, we use a second string type, `String`. This type is allocated on the heap and as such can grow in size. You can create a `String` object using the `from` function, like so:

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

The double colon (`::`) is an operator that allows us to namespace this particular function under the `String` type rather than using some sort of name like `string_from`. We'll learn more about namespaces and the syntax more in the “Method Syntax” section of Chapter 5 and when we talk about modules and crates in “Module Definitions” in Chapter 7.

This kind of string *can* be mutated:

```
let mut s = String::from("hello");
s.push_str(", world!"); // push_str() appends a literal to a String
println!("{}", s); // This will print `hello, world!`
```

So, what's the difference here? Why can `String` be mutated but literals can't? Let's look at how these two types deal with memory.

Memory and Allocation

In the case of a string literal, we know the contents at compile time, so the text is copied directly into the final executable. This is why string literals are fast and efficient. But what about strings whose contents come from the string literal's immutability. Unfortunately, we can't put a blob of memory for each piece of text whose size is unknown at compile time and whose contents change while running the program.

With the `String` type, in order to support a mutable, growable piece of text, we need to allocate a certain amount of memory on the heap, unknown at compile time, to hold the contents of the string.

- The memory must be requested from the operating system at runtime
- We need a way of returning this memory to the operating system when we're done with it.

That first part is done by us: when we call `String::from`, its implementation needs to request memory from the OS. This is pretty much universal in programming languages.

However, the second part is different. In languages with a *garbage collector* (like Java), the GC automatically cleans up memory that isn't being used anymore, and we don't need to worry about it. In Rust, it's our responsibility to identify when memory is no longer being used and return it, just as we did to request it. Doing this correctly has historically been a difficult programming problem. If we forget, we'll waste memory. If we do it too early, we'll leak memory. If we do it twice, that's a bug too. We need to pair exactly one `alloc` with one `free`.

Rust takes a different path: the memory is automatically returned once the variable goes out of scope. Here's a version of our scope example from Listing 4-1 using a string literal:

```
{  
    let s = String::from("hello"); // s is valid from this point forward  
    // do stuff with s  
}  
// this scope is now over, and s is no longer valid
```

There is a natural point at which we can return the memory our `String` needs to the system: when `s` goes out of scope. When a variable goes out of scope, Rust calls a function called `drop`, and it's where the author of `String` can tell the system to free the memory. Rust calls `drop` automatically at the closing `}`.

Note: In C++, this pattern of deallocating resources at the end of an item's scope is sometimes called *Resource Acquisition Is Initialization (RAII)*. The `drop` function is similar to the `destructor` you might be familiar with if you've used RAII patterns.

This pattern has a profound impact on the way Rust code is written. It may seem like common sense, but the behavior of code can be unexpected in more complicated situations when multiple variables use the data we've allocated on the heap. Let's explore some examples of this behavior now.

Ways Variables and Data Interact: Move

Multiple variables can interact with the same data in different ways in Rust. I'll show you how this works using an integer in Listing 4-2:

```
let x = 5;
let y = x;
```

Listing 4-2: Assigning the integer value of variable `x` to `y`

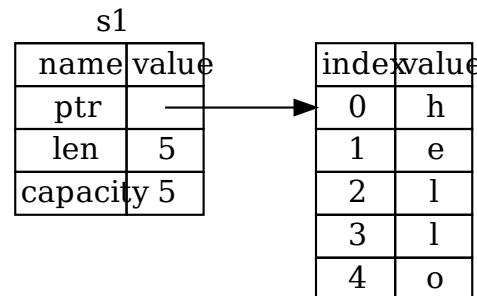
We can probably guess what this is doing: “bind the value `5` to `x`; then make `x` and bind it to `y`.” We now have two variables, `x` and `y`, and both equal because integers are simple values with a known, fixed size, and pushed onto the stack.

Now let’s look at the `String` version:

```
let s1 = String::from("hello");
let s2 = s1;
```

This looks very similar to the previous code, so we might assume that the was the same: that is, the second line would make a copy of the value in `s1` and bind it to `s2`. But quite what happens.

Take a look at Figure 4-1 to see what is happening to `String` under the covers. It is made up of three parts, shown on the left: a pointer to the memory that holds the length, a capacity, and a pointer to the heap that holds the contents. On the right is the representation of the string “hello” on the heap.

Figure 4-1: Representation in memory of a `String` holding the value “hello”

The length is how much memory, in bytes, the contents of the `String` is currently using. The capacity is the total amount of memory, in bytes, that the `String` has received from the system. The difference between length and capacity matters, but not in this case; it is fine to ignore the capacity.

When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the length, capacity, and pointer to the heap that holds the contents. We do not copy the data on the heap that “hello”. In other words, the data representation in memory looks like Figure 4-2.

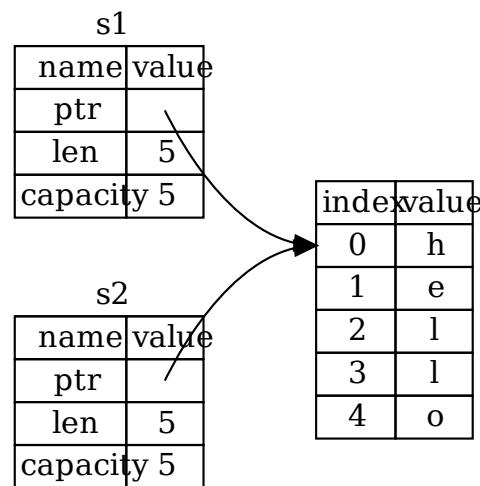


Figure 4-2: Representation in memory of the variable `s2` that has a copy of `s1`

The representation does *not* look like Figure 4-3, which is what memory would instead copied the heap data as well. If Rust did this, the operation `s2 = s1` in terms of runtime performance if the data on the heap were large.

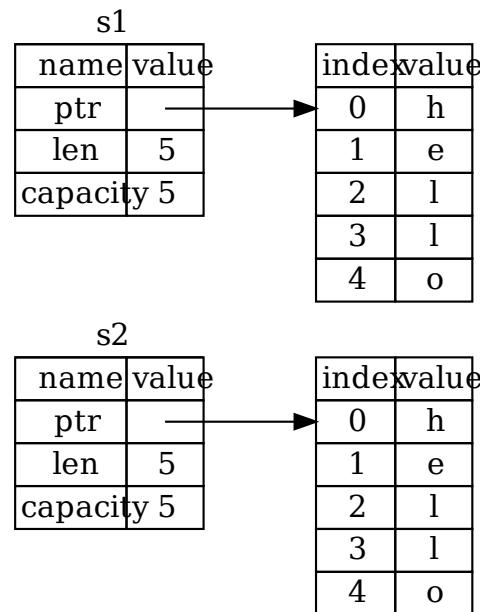


Figure 4-3: Another possibility for what `s2 = s1` might do if Rust copied the

Earlier, we said that when a variable goes out of scope, Rust automatically copies and cleans up the heap memory for that variable. But Figure 4-2 shows both pointers to the same location. This is a problem: when `s2` and `s1` go out of scope, they both free the same memory. This is known as a *double free* error and is one of the most common errors mentioned previously. Freeing memory twice can lead to memory corruption or lead to security vulnerabilities.

To ensure memory safety, there's one more detail to what happens in this situation:

of trying to copy the allocated memory, Rust considers `s1` to no longer be valid. It doesn't need to free anything when `s1` goes out of scope. Check out what happens if we try to use `s1` after `s2` is created; it won't work:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{} world!", s1);
```

You'll get an error like this because Rust prevents you from using the invalid variable:

```
error[E0382]: use of moved value: `s1`
--> src/main.rs:5:28
   |
3 |     let s2 = s1;
   |         -- value moved here
4 |
5 |     println!("{} world!", s1);
   |             ^^^ value used here after move
   |
= note: move occurs because `s1` has type `std::string::String`,
not implement the `Copy` trait
```

If you've heard the terms *shallow copy* and *deep copy* while working with other languages, this is the concept of copying the pointer, length, and capacity without copying the data itself, resulting in a shallow copy. But because Rust also invalidates the first variable, it's known as a *move*. Here we would read this by saying that `s1` was moved. So what actually happens is shown in Figure 4-4.

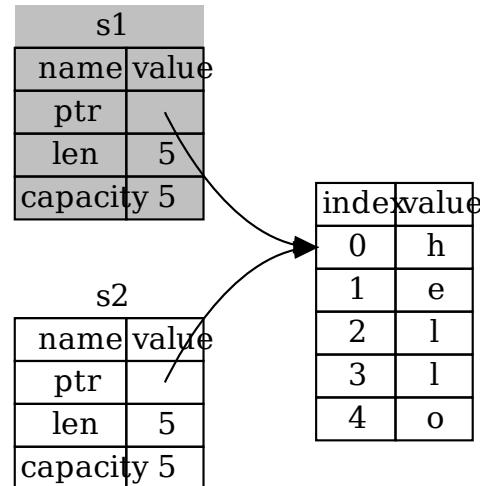


Figure 4-4: Representation in memory after `s1` has been invalidated

That solves our problem! With only `s2` valid, when it goes out of scope, it automatically gets deallocated, and we're done.

In addition, there's a design choice that's implied by this: Rust will never automatically copy the heap-allocated memory of your data. Therefore, any *automatic* copying can be assumed to be a *move*, which is good for runtime performance.

Ways Variables and Data Interact: Clone

If we *do* want to deeply copy the heap data of the `String`, not just the stack pointer, we can do so by implementing the `Clone` trait:

common method called `clone`. We'll discuss method syntax in Chapter 5, but it's a common feature in many programming languages, you've probably seen it before.

Here's an example of the `clone` method in action:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

This works just fine and explicitly produces the behavior shown in Figure 4-3: `s1` does not get copied.

When you see a call to `clone`, you know that some arbitrary code is being executed. It may be expensive. It's a visual indicator that something different is going on.

Stack-Only Data: Copy

There's another wrinkle we haven't talked about yet. This code using integers, which we showed earlier in Listing 4-2, works and is valid:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

But this code seems to contradict what we just learned: we don't have a call to `clone`, `x` is valid and wasn't moved into `y`.

The reason is that types such as integers that have a known size at compile time are stored on the stack, so copies of the actual values are quick to make. That means the compiler would want to prevent `x` from being valid after we create the variable `y`. In fact, there's a difference between deep and shallow copying here, so calling `clone` wouldn't change anything from the usual shallow copying and we can leave it out.

Rust has a special annotation called the `Copy` trait that we can place on types that are stored on the stack (we'll talk more about traits in Chapter 10). If a type has the `Copy` trait, its variable is still usable after assignment. Rust won't let us annotate a type with the `Copy` trait if the type needs to be dropped, or if any of its parts, has implemented the `Drop` trait. If the type needs to be dropped, the compiler will issue an error. To learn about how to add the `Copy` annotation to your types, see "Adding Traits" in Appendix C.

So what types are `Copy`? You can check the documentation for the given type. In general, any group of simple scalar values can be `Copy`, and nothing that contains some form of resource is `Copy`. Here are some of the types that are `Copy`:

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, but only if they contain types that are also `Copy`. For example, `(i32, String)` is not.

Ownership and Functions

The semantics for passing a value to a function are similar to those for assignment. Passing a variable to a function will move or copy, just as assignment does. Consider an example with some annotations showing where variables go into and out of scope.

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello"); // s comes into scope
    takes_ownership(s);          // s's value moves into the function
                                 // ... and so is no longer valid
    let x = 5;                   // x comes into scope
    makes_copy(x);              // x would move into the function
                                // but i32 is Copy, so it's ok
                                // to use x afterward
} // Here, x goes out of scope, then s. But because s's value was moved,
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Listing 4-3: Functions with ownership and scope annotated

If we tried to use `s` after the call to `takes_ownership`, Rust would throw a compilation error. Rust's static checks protect us from mistakes. Try adding code to `main` that uses `s` and `x` after they go out of scope to see what happens.

Return Values and Scope

Returning values can also transfer ownership. Listing 4-4 is an example with the same code as Listing 4-3, but with a return value.

Filename: src/main.rs

```

fn main() {
    let s1 = gives_ownership();          // gives_ownership moves it
                                         // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                         // takes_and_gives_back, which
                                         // moves its return value
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope
  // moved, so nothing happens. s1 goes out of scope and is dropped

fn gives_ownership() -> String {           // gives_ownership will
                                             // return value into s1
                                             // that calls it

    let some_string = String::from("hello"); // some_string comes into
                                              // scope

    some_string                           // some_string is returned
                                         // moves out to the calling
                                         // function.
}

// takes_and_gives_back will take a String and return one.
fn takes_and_gives_back(a_string: String) -> String { // a_string comes
                                                       // into scope

    a_string // a_string is returned and moves out to the calling
}

```

Listing 4-4: Transferring ownership of return values

The ownership of a variable follows the same pattern every time: assigning a variable moves it. When a variable that includes data on the heap goes out of scope, it's cleaned up by `drop` unless the data has been moved to be owned by another variable.

Taking ownership and then returning ownership with every function is a bit tedious. Is there a way to let a function use a value but not take ownership? It's quite annoying that we need to be passed back if we want to use it again, in addition to any data returned by the function that we might want to return as well.

It's possible to return multiple values using a tuple, as shown in Listing 4-5:

Filename: src/main.rs

```

fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}

```

Listing 4-5: Returning ownership of parameters

But this is too much ceremony and a lot of work for a concept that should be handled automatically for us. Rust has a feature for this concept, called *references*.

References and Borrowing

The issue with the tuple code in Listing 4-5 is that we have to return the `str` function so we can still use the `String` after the call to `calculate_length`, I moved into `calculate_length`.

Here is how you would define and use a `calculate_length` function that has an object as a parameter instead of taking ownership of the value:

Filename: `src/main.rs`

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

First, notice that all the tuple code in the variable declaration and the function definition is gone. Second, note that we pass `&s1` into `calculate_length` and, in its definition, we pass `s` rather than `String`.

These ampersands are *references*, and they allow you to refer to some value without taking ownership of it. Figure 4-5 shows a diagram.

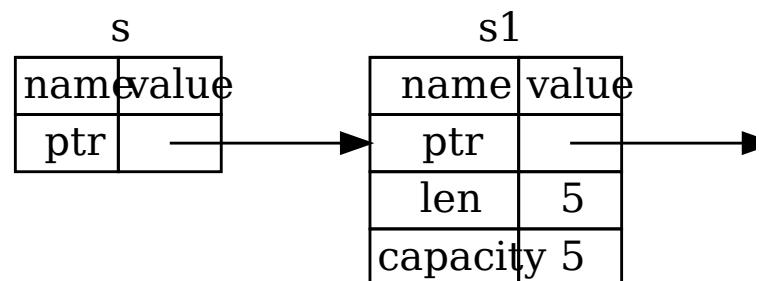


Figure 4-5: A diagram of `&String s` pointing at `String s1`

Note: The opposite of referencing by using `&` is *dereferencing*, which is achieved by using the dereference operator, `*`. We'll see some uses of the dereference operator and discuss details of dereferencing in Chapter 15.

Let's take a closer look at the function call here:

```
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

The `&s1` syntax lets us create a reference that *refers* to the value of `s1` but it does not own it, the value it points to will not be dropped when the reference goes out of scope.

Likewise, the signature of the function uses `&` to indicate that the type of the reference. Let's add some explanatory annotations:

```
fn calculate_length(s: &String) -> usize { // s is a reference to s
    s.len()
} // Here, s goes out of scope. But because it does not have ownership,
  // it refers to, nothing happens.
```

The scope in which the variable `s` is valid is the same as any function parameter: it doesn't drop what the reference points to when it goes out of scope because we never had ownership of it.

When functions have references as parameters instead of the actual values, we're giving them the values in order to give back ownership, because we never had ownership of them.

We call having references as function parameters *borrowing*. As in real life, if you borrow something, you can use it but you can't change it. When you're done, you have to give it back.

So what happens if we try to modify something we're borrowing? Try the code from Listing 4-6. What do you think will happen? Hint: it doesn't work!

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Listing 4-6: Attempting to modify a borrowed value

Here's the error:

```
error[E0596]: cannot borrow immutable borrowed content `*some_string` as mutable
--> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |           ----- use `&mut String` here to make it mutable
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^ cannot borrow as mutable
```

Just as variables are immutable by default, so are references. We're not allowed to change the value that a reference points to. That's why we have a reference to.

Mutable References

We can fix the error in the code from Listing 4-6 with just a small tweak:

Filename: src/main.rs

```

fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}

```

First, we had to change `s` to be `mut`. Then we had to create a mutable reference to accept a mutable reference with `some_string: &mut String`.

But mutable references have one big restriction: you can only have one mutable reference to a particular piece of data in a particular scope. This code will fail:

Filename: src/main.rs

```

let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

```

Here's the error:

```

error[E0499]: cannot borrow `s` as mutable more than once at a time
--> borrow_twice.rs:5:19
 |
4 |     let r1 = &mut s;
|             - first mutable borrow occurs here
5 |     let r2 = &mut s;
|             ^ second mutable borrow occurs here
6 | }
| - first borrow ends here

```

This restriction allows for mutation but in a very controlled fashion. It's something that Rustaceans struggle with, because most languages let you mutate whenever you want.

The benefit of having this restriction is that Rust can prevent data races at compile time. This is similar to a race condition and happens when these three behaviors occur simultaneously:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix. Rust prevents them from happening at runtime; Rust prevents this problem from happening by preventing the code with data races!

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not *simultaneous* ones:

```

let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with it

let r2 = &mut s;

```

A similar rule exists for combining mutable and immutable references. This is covered in the next section.

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
```

Here's the error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
--> borrow_thrice.rs:6:19
   |
4 |     let r1 = &s; // no problem
   |             - immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
   |             ^ mutable borrow occurs here
7 | }
   | - immutable borrow ends here
```

Whew! We *also* cannot have a mutable reference while we have an immutable reference don't expect the values to suddenly change out from under us. Multiple immutable references are okay because no one who is just reading to affect anyone else's reading of the data.

Even though these errors may be frustrating at times, remember that it's better to catch out a potential bug early (at compile time rather than at runtime) and show it to the person who wrote the code. Then you don't have to track down why your data isn't what you expected.

Dangling References

In languages with pointers, it's easy to erroneously create a *dangling pointer*, which is a location in memory that may have been given to someone else, but no longer has valid memory while preserving a pointer to that memory. In Rust, by contrast, the language ensures that references will never be dangling references: if you have a reference to something, the language will ensure that the data will not go out of scope before the reference to the data does.

Let's try to create a dangling reference, which Rust will prevent with a compilation error.

Filename: src/main.rs

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Here's the error:

```
error[E0106]: missing lifetime specifier
--> dangle.rs:5:16
|
5 | fn dangle() -> &String {
|           ^ expected lifetime parameter
|
|= help: this function's return type contains a borrowed value, but
no value for it to be borrowed from
|= help: consider giving it a 'static lifetime
```

This error message refers to a feature we haven't covered yet: *lifetimes*. We'll detail in Chapter 10. But, if you disregard the parts about lifetimes, the message key to why this code is a problem:

```
this function's return type contains a borrowed value, but there is
for it to be borrowed from.
```

Let's take a closer look at exactly what's happening at each stage of our `dangle`:

```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away!
// Danger!
```

Because `s` is created inside `dangle`, when the code of `dangle` is finished, `s` is dropped. But we tried to return a reference to it. That means this reference would be pointing to freed memory. That's no good! Rust won't let us do this.

The solution here is to return the `String` directly:

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

This works without any problems. Ownership is moved out, and nothing is dropped.

The Rules of References

Let's recap what we've discussed about references:

- At any given time, you can have *either* (but not both of) one mutable reference or immutable references.
- References must always be valid.

Next, we'll look at a different kind of reference: slices.

The Slice Type

Another data type that does not have ownership is the *slice*. Slices let you refer to a sequence of elements in a collection rather than the whole collection.

Here's a small programming problem: write a function that takes a string and

it finds in that string. If the function doesn't find a space in the string, the whole word, so the entire string should be returned.

Let's think about the signature of this function:

```
fn first_word(s: &String) -> ?
```

This function, `first_word`, has a `&String` as a parameter. We don't want ownership of the string. But what should we return? We don't really have a way to talk about *part* of a string. We could return the index of the end of the word. Let's try that, as shown in Listing 4-7.

Filename: `src/main.rs`

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Listing 4-7: The `first_word` function that returns a byte index value into the string.

Because we need to go through the `String` element by element and check for a space, we'll convert our `String` to an array of bytes using the `as_bytes` method:

```
let bytes = s.as_bytes();
```

Next, we create an iterator over the array of bytes using the `iter` method:

```
for (i, &item) in bytes.iter().enumerate() {
```

We'll discuss iterators in more detail in Chapter 13. For now, know that `iter` returns each element in a collection and that `enumerate` wraps the result of each element as part of a tuple instead. The first element of the tuple returned from `enumerate` is the index, and the second element is a reference to the element. This is a bit more work than calculating the index ourselves.

Because the `enumerate` method returns a tuple, we can use patterns to describe it everywhere else in Rust. So in the `for` loop, we specify a pattern that has `i` for the index and `&item` for the single byte in the tuple. Because we get a reference to the byte from `bytes.iter().enumerate()`, we use `&` in the pattern.

Inside the `for` loop, we search for the byte that represents the space by using the `==` syntax. If we find a space, we return the position. Otherwise, we return the length of the string using `s.len()`:

```
if item == b' ' {
    return i;
}
s.len()
```

We now have a way to find out the index of the end of the first word in the string. We're returning a `usize` on its own, but it's only a meaningful number if there's a space at the end of the word.

the `&string`. In other words, because it's a separate value from the `String` that it will still be valid in the future. Consider the program in Listing 4-8 that function from Listing 4-7:

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5

    s.clear(); // This empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string
    // we could meaningfully use the value 5 with. word is now tot;
}
```

Listing 4-8: Storing the result from calling the `first_word` function and then contents

This program compiles without any errors and would also do so if we used `s.clear()`. Because `word` isn't connected to the state of `s` at all, `word` still We could use that value `5` with the variable `s` to try to extract the first word a bug because the contents of `s` have changed since we saved `5` in `word`.

Having to worry about the index in `word` getting out of sync with the data in prone! Managing these indices is even more brittle if we write a `second_word` would have to look like this:

```
fn second_word(s: &String) -> (usize, usize) {
```

Now we're tracking a starting *and* an ending index, and we have even more \ calculated from data in a particular state but aren't tied to that state at all. W unrelated variables floating around that need to be kept in sync.

Luckily, Rust has a solution to this problem: string slices.

String Slices

A *string slice* is a reference to part of a `String`, and it looks like this:

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

This is similar to taking a reference to the whole `String` but with the extra a reference to the entire `String`, it's a reference to a portion of the `String` is a range that begins at `start` and continues up to, but not including, `end`. `end`, we can use `..=` instead of `..`:

```
let s = String::from("hello world");

let hello = &s[0..=4];
let world = &s[6..=10];
```

The `=` means that we're including the last number, if that helps you rememl

between `..` and `..=`.

We can create slices using a range within brackets by specifying `[starting_index .. ending_index]`, where `starting_index` is the first position in the slice and `ending_index` is the last position in the slice. Internally, the slice data structure stores the starting pointer to the slice, which corresponds to `ending_index minus starting_index`. So if we write `let world = &s[6..11];`, `world` would be a slice that contains a pointer to the memory starting at index 6 with a length value of 5.

Figure 4-6 shows this in a diagram.

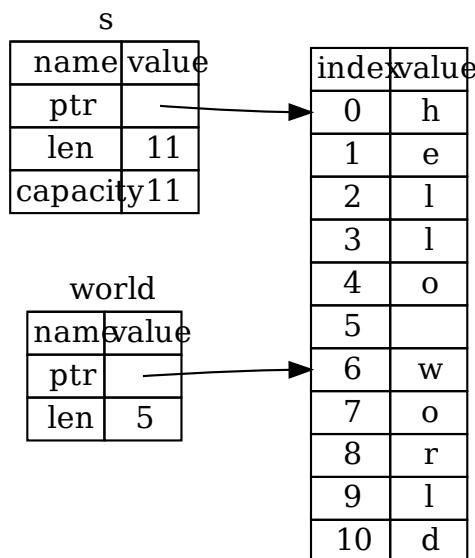


Figure 4-6: String slice referring to part of a `String`

With Rust's `..` range syntax, if you want to start at the first index (zero), you can drop the `0` before the two periods. In other words, these are equal:

```
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[..2];
```

By the same token, if your slice includes the last byte of the `String`, you can drop the `len` number. That means these are equal:

```
let s = String::from("hello");
let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];
```

You can also drop both values to take a slice of the entire string. So these are equal:

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

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

Note: String slice range indices must occur at valid UTF-8 character bounds to create a string slice in the middle of a multibyte character, your program will error. For the purposes of introducing string slices, we are assuming ASCII. A more thorough discussion of UTF-8 handling is in the “Strings” section of the book.

With all this information in mind, let’s rewrite `first_word` to return a slice. Instead of “string slice” is written as `&str`:

Filename: src/main.rs

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

We get the index for the end of the word in the same way as we did in Listing 1, but instead of returning a string, we return a slice. When we find the first occurrence of a space, we return a string slice up to and including the index of the space as the starting and ending indices.

Now when we call `first_word`, we get back a single value that is tied to the string. The value is made up of a reference to the starting point of the slice and the number of bytes in the slice.

Returning a slice would also work for a `second_word` function:

```
fn second_word(s: &String) -> &str {
```

We now have a straightforward API that’s much harder to mess up, because the references into the `String` remain valid. Remember the bug in the program from Listing 1? We got the index to the end of the first word but then cleared the string so our code would panic. That code was logically incorrect but didn’t show any immediate errors. The problem would have come up later if we kept trying to use the first word index with an emptied string. It’s impossible to make up for this mistake and let us know we have a problem with our code much sooner. If we call `first_word` on an empty string, the function will throw a compile time error:

Filename: src/main.rs

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // Error!
}
```

Here's the compiler error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
--> src/main.rs:6:5
  |
4 |     let word = first_word(&s);
  |             ^ immutable borrow occurs here
5 |
6 |     s.clear(); // Error!
  |         ^ mutable borrow occurs here
7 | }
  | - immutable borrow ends here
```

Recall from the borrowing rules that if we have an immutable reference to `s`, we can't also take a mutable reference. Because `clear` needs to truncate the `String`, it must take a mutable reference, which fails. Not only has Rust made our API easier to use, but it has also eliminated an entire class of errors at compile time!

String Literals Are Slices

Recall that we talked about string literals being stored inside the binary. Now that we know what slices are, we can properly understand string literals:

```
let s = "Hello, world!";
```

The type of `s` here is `&str`: it's a slice pointing to that specific point of the binary. String literals are immutable; `&str` is an immutable reference.

String Slices as Parameters

Knowing that you can take slices of literals and `String s` leads us to one more question: what does `first_word` do? And that's its signature:

```
fn first_word(s: &String) -> &str {
```

A more experienced Rustacean would write the following line instead because it's more idiomatic: it takes a slice of `String` instead of a reference to both `String` and `&str`:

```
fn first_word(s: &str) -> &str {
```

If we have a string slice, we can pass that directly. If we have a `String`, we can convert it to a slice. Defining a function to take a string slice instead of a reference to a `String` makes our API more general and useful without losing any functionality:

Filename: `src/main.rs`

```
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Other Slices

String slices, as you might imagine, are specific to strings. But there's a more general way to work with slices.

Consider this array:

```
let a = [1, 2, 3, 4, 5];
```

Just as we might want to refer to a part of a string, we might want to refer to a part of an array. To do so like this:

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

This slice has the type `&[i32]`. It works the same way as string slices do, by pointing to the first element and a length. You'll use this kind of slice for all sorts of other things throughout the book. We'll discuss these collections in detail when we talk about vectors in Chapter 8.

Summary

The concepts of ownership, borrowing, and slices ensure memory safety in Rust at compile time. The Rust language gives you control over your memory usage that you don't have in other systems programming languages, but having the owner of data automatically drop it when the owner goes out of scope means you don't have to write and control this manually.

Ownership affects how lots of other parts of Rust work, so we'll talk about them throughout the rest of the book. Let's move on to Chapter 5 and look at how we can group related data together in a `struct`.

Using Structs to Structure Related Data

A `struct`, or `structure`, is a custom data type that lets you name and package together values that make up a meaningful group. If you're familiar with an object-oriented language, a `struct` is like an object's data attributes. In this chapter, we'll compare and contrast `structs` with `enums`, demonstrate how to use structs, and discuss how to define methods and associated functions to specify behavior associated with a struct's data. Structs and enums (discussed in Chapter 4) are the most common ways to structure data in Rust.

building blocks for creating new types in your program's domain to take full compile time type checking.

Defining and Instantiating Structs

Structs are similar to tuples, which were discussed in Chapter 3. Like tuples, can be different types. Unlike with tuples, you'll name each piece of data so I mean. As a result of these names, structs are more flexible than tuples; you order of the data to specify or access the values of an instance.

To define a struct, we enter the keyword `struct` and name the entire struct describe the significance of the pieces of data being grouped together. Then we define the names and types of the pieces of data, which we call *fields*. For shows a struct that stores information about a user account:

```
struct User {  
    username: String,  
    email: String,  
    sign_in_count: u64,  
    active: bool,  
}
```

Listing 5-1: A `User` struct definition

To use a struct after we've defined it, we create an *instance* of that struct by : values for each of the fields. We create an instance by stating the name of the curly brackets containing `key: value` pairs, where the keys are the names of the values are the data we want to store in those fields. We don't have to specify the order in which we declared them in the struct. In other words, the struct defines template for the type, and instances fill in that template with particular data type. For example, we can declare a particular user as shown in Listing 5-2:

```
let user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};
```

Listing 5-2: Creating an instance of the `User` struct

To get a specific value from a struct, we can use dot notation. If we wanted just the address, we could use `user1.email` wherever we wanted to use this value. If the struct is mutable, we can change a value by using the dot notation and assigning into it. Listing 5-3 shows how to change the value in the `email` field of a mutable `User` instance.

```
let mut user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};  
  
user1.email = String::from("anotheremail@example.com");
```

Listing 5-3: Changing the value in the `email` field of a `User` instance

Note that the entire instance must be mutable; Rust doesn't allow us to make a struct mutable.

As with any expression, we can construct a new instance of the struct as the last statement in the function body to implicitly return that new instance. Listing 5-4 shows a `build_user` function that returns a `User` instance with the given email and username. The `active` field is set to `true`, and the `sign_in_count` gets a value of `1`.

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

Listing 5-4: A `build_user` function that takes an email and username and returns a `User` instance.

It makes sense to name the function parameters with the same name as the struct fields, so we can repeat the `email` and `username` field names and variables is a bit tedious. If we had many fields, repeating each name would get even more annoying. Luckily, there's a better way.

Using the Field Init Shorthand when Variables and Fields Have the Same Name

Because the parameter names and the struct field names are exactly the same, we can use the *field init shorthand* syntax to rewrite `build_user` so that it behaves exactly like Listing 5-4, but without having to repeat the `email` and `username` as shown in Listing 5-5.

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Listing 5-5: A `build_user` function that uses field init shorthand because the function parameters have the same name as struct fields

Here, we're creating a new instance of the `User` struct, which has a field named `email`. We can set the `email` field's value to the value in the `email` parameter of the `build_user` function. Because the `email` field and the `email` parameter have the same name, we can just write `email` rather than `email: email`.

Creating Instances From Other Instances With Struct Update

It's often useful to create a new instance of a struct that uses most of an old instance but changes some. You'll do this using *struct update syntax*.

First, Listing 5-6 shows how we create a new `User` instance in `user2` without setting new values for `email` and `username` but otherwise use the same values created in Listing 5-2:

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

Listing 5-6: Creating a new `User` instance using some of the values from `user1`.

Using struct update syntax, we can achieve the same effect with less code, a lot less! The syntax `..` specifies that the remaining fields not explicitly set should have the same values as the fields in the given instance.

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

Listing 5-7: Using struct update syntax to set new `email` and `username` values but use the rest of the values from the fields of the instance in the `user1` variable.

The code in Listing 5-7 also creates an instance in `user2` that has a different `username` but has the same values for the `active` and `sign_in_count` fields.

Tuple Structs without Named Fields to Create Different Types

You can also define structs that look similar to tuples, called *tuple structs*. Tuple structs add meaning the struct name provides but don't have names associated with them; they just have the types of the fields. Tuple structs are useful when you want to give a name and make the tuple be a different type than other tuples, and naming it a regular struct would be verbose or redundant.

To define a tuple struct start with the `struct` keyword and the struct name followed by the tuple. For example, here are definitions and usages of two tuple structs, `Color` and `Point`:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Note that the `black` and `origin` values are different types, because they're tuple structs. Each struct you define is its own type, even though the fields within them are the same types. For example, a function that takes a parameter of type `Color` can take a `Color` tuple as an argument, even though both types are made up of three `i32` values. Other tuple struct instances behave like tuples: you can destructure them into their individual fields using `.0`, `.1`, and so on, or followed by the index to access an individual value, and so on.

Unit-Like Structs Without Any Fields

You can also define structs that don't have any fields! These are called *unit-like* structs because they behave similarly to `()`, the unit type. Unit-like structs can be useful in situations where you want to represent a value that has no data but needs to be part of a larger type.

to implement a trait on some type but don't have any data that you want to own. We'll discuss traits in Chapter 10.

Ownership of Struct Data

In the `User` struct definition in Listing 5-1, we used the owned `String` type instead of the `&str` string slice type. This is a deliberate choice because we want instances of `User` to own all of its data and for that data to be valid for as long as the entire struct.

It's possible for structs to store references to data owned by something else. Rust requires the use of *lifetimes*, a Rust feature that we'll discuss in Chapter 10, to ensure that the data referenced by a struct is valid for as long as the struct is. Let's look at a reference in a struct without specifying lifetimes, like this, which won't work:

Filename: `src/main.rs`

```
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

The compiler will complain that it needs lifetime specifiers:

```
error[E0106]: missing lifetime specifier
-->
|
2 |     username: &str,
|           ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
-->
|
3 |     email: &str,
|           ^ expected lifetime parameter
```

In Chapter 10, we'll discuss how to fix these errors so you can store references to owned data. For now, we'll fix errors like these using owned types like `String` instead of `&str`.

An Example Program Using Structs

To understand when we might want to use structs, let's write a program that calculates the area of a rectangle. We'll start with single variables, and then refactor the program using structs instead.

Let's make a new binary project with Cargo called `rectangles` that will take the width and height of a rectangle as command-line arguments and print the area.

rectangle specified in pixels and calculate the area of the rectangle. Listing 5 shows a program with one way of doing exactly that in our project's `src/main.rs`:

Filename: `src/main.rs`

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Listing 5-8: Calculating the area of a rectangle specified by separate width and height.

Now, run this program using `cargo run`:

```
The area of the rectangle is 1500 square pixels.
```

Even though Listing 5-8 works and figures out the area of the rectangle by calculating the product of width and height separately, we can do better. The width and the height are related because together they describe one rectangle.

The issue with this code is evident in the signature of `area`:

```
fn area(width: u32, height: u32) -> u32 {
```

The `area` function is supposed to calculate the area of one rectangle, but the signature lists two parameters. The parameters are related, but that's not expressed anywhere in the code. It would be more readable and more manageable to group width and height together. We'll discuss one way we might do that in "The Tuple Type" section of Chapter 3.

Refactoring with Tuples

Listing 5-9 shows another version of our program that uses tuples:

Filename: `src/main.rs`

```
fn main() {
    let rect1 = (30, 50);

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

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Listing 5-9: Specifying the width and height of the rectangle with a tuple

In one way, this program is better. Tuples let us add a bit of structure, and we can add more arguments. But in another way, this version is less clear: tuples don't have names.

calculation has become more confusing because we have to index into the `width` and `height` fields.

It doesn't matter if we mix up width and height for the area calculation, but it would matter if we did this for a rectangle on the screen, it would matter! We would have to keep in mind that `width` is the tuple index `0` and `height` is the tuple index `1`. If someone else worked on this code, they would have to figure this out and keep it in mind as well. It would be easy to forget or mix up the indices, which could cause errors, because we haven't conveyed the meaning of our data in our code.

Refactoring with Structs: Adding More Meaning

We use structs to add meaning by labeling the data. We can transform the tuple `(width, height)` into a data type with a name for the whole as well as names for the parts, as shown in Listing 5-10.

Filename: `src/main.rs`

```
struct Rectangle {
    width: u32,
    height: u32,
}

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

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

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

Listing 5-10: Defining a `Rectangle` struct

Here we've defined a struct and named it `Rectangle`. Inside the curly braces, we've given it two fields, `width` and `height`, both of which have type `u32`. Then in `main`, we create a variable `rect1` of type `Rectangle` that has a width of 30 and a height of 50.

Our `area` function is now defined with one parameter, which we've named `rectangle`. This is an immutable borrow of a `Rectangle` instance. As mentioned in Chapter 4, we use `&` to borrow the struct rather than take ownership of it. This way, `main` retains its ownership of `rect1` and can continue using `rect1`, which is the reason we use the `&` in the function signature of `area`.

The `area` function accesses the `width` and `height` fields of the `Rectangle` struct. The signature for `area` now says exactly what we mean: calculate the area of `Rectangle`. By giving the function a descriptive name, we're conveying meaning to the values rather than using the tuple index values `0` and `1` for clarity.

Adding Useful Functionality with Derived Traits

It'd be nice to be able to print an instance of `Rectangle` while we're debugging. We can do this by printing the values for all its fields. Listing 5-11 tries using the `println!` macro as we did in previous chapters. This won't work, however:

Filename: `src/main.rs`

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
fn main() {  
    let rect1 = Rectangle { width: 30, height: 50 };  
  
    println!("rect1 is {}", rect1);  
}
```

Listing 5-11: Attempting to print a `Rectangle` instance

When we run this code, we get an error with this core message:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not
```

The `println!` macro can do many kinds of formatting, and by default, curly to use formatting known as `Display`: output intended for direct end user primitive types we've seen so far implement `Display` by default, because they want to show a `1` or any other primitive type to a user. But with structs, the format the output is less clear because there are more display possibilities: `{}?`? `:{}`? Do you want to print the curly brackets? Should all the fields be shown? Rust doesn't try to guess what we want, and structs don't have a provided implementation for `Display`.

If we continue reading the errors, we'll find this helpful note:

```
'Rectangle' cannot be formatted with the default formatter; try using  
'{:?}' instead if you are using a format string
```

Let's try it! The `println!` macro call will now look like `println!("rect1 is {:?}", rect1)`. The specifier `:?`` inside the curly brackets tells `println!` we want to use an `Debug` trait. `Debug` is a trait that enables us to print our struct in a way that is useful for us to see its value while we're debugging our code.

Run the code with this change. Drat! We still get an error:

```
error[E0277]: the trait bound `Rectangle: std::fmt::Debug` is not
```

But again, the compiler gives us a helpful note:

```
'Rectangle' cannot be formatted using `{:?}`; if it is defined in your  
crate, add `#[derive(Debug)]` or manually implement it
```

Rust *does* include functionality to print out debugging information, but we have to make that functionality available for our struct. To do that, we add the annotation `#[derive(Debug)]` just before the struct definition, as shown in Listing 5-12.

Filename: `src/main.rs`

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

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

    println!("rect1 is {:?}", rect1);
}
```

Listing 5-12: Adding the annotation to derive the `Debug` trait and printing the struct using debug formatting

Now when we run the program, we won't get any errors, and we'll see the following output:

```
rect1 is Rectangle { width: 30, height: 50 }
```

Nice! It's not the prettiest output, but it shows the values of all the fields for the struct. This kind of output would definitely help during debugging. When we have larger structs, it's useful to use the `{:#?}` style instead of `{:?:}` in the `println!` macro. If we use the `{:#?}` style in the example, the output will look like this:

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

Rust has provided a number of traits for us to use with the `derive` annotation to add specific behavior to our custom types. Those traits and their behaviors are listed in [Appendix A: Traits](#). We'll cover how to implement these traits with custom behavior as well as how to define our own traits in Chapter 10.

Our `area` function is very specific: it only computes the area of rectangles. It's not a good idea to tie this behavior more closely to our `Rectangle` struct, because it won't work well with other shapes. Instead, we can look at how we can continue to refactor this code by turning the `area` function into a method defined on our `Rectangle` type.

Method Syntax

Methods are similar to functions: they're declared with the `fn` keyword and have parameters and a return value, and they contain some code that is run from somewhere else. However, methods are different from functions in that they are defined in the context of a struct (or an enum or a trait object, which we cover in Chapters 10, 11, and 12 respectively), and their first parameter is always `self`, which represents the instance of the struct on which the method is being called on.

Defining Methods

Let's change the `area` function that has a `Rectangle` instance as a parameter into a 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` (ir
Then we move the `area` function within the `impl` curly brackets and chang
case, only) parameter to be `self` in the signature and everywhere within th
we called the `area` function and passed `rect1` as an argument, we can inst
to call the `area` method on our `Rectangle` instance. The method syntax go
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 type of `self` is `Rectangle` due to this method's being inside the `impl` F
that we still need to use the `&` before `self`, just as we did in `&Rectangle`. M
ownership of `self`, borrow `self` immutably as we've done here, or borrow
they can any other parameter.

We've chosen `&self` here for the same reason we used `&Rectangle` in the i
don't want to take ownership, and we just want to read the data in the struc
wanted to change the instance that we've called the method on as part of w
we'd use `&mut self` as the first parameter. Having a method that takes owr
by using just `self` as the first parameter is rare; this technique is usually us
transforms `self` into something else and you want to prevent the caller fro
instance after the transformation.

The main benefit of using methods instead of functions, in addition to using
having to repeat the type of `self` in every method's signature, is for organiz
things we can do with an instance of a type in one `impl` block rather than m
our code search for capabilities of `Rectangle` in various places in the library

Where's the `->` Operator?

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

Rust doesn't have an equivalent to the `->` operator; instead, Rust has a fe
automatic referencing and dereferencing. Calling methods is one of the few

has this behavior.

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

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

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

Methods with More Parameters

Let's practice using methods by implementing a second method on the `Rect` we want an instance of `Rectangle` to take another instance of `Rectangle` a second `Rectangle` can fit completely within `self`; otherwise it should retur want to be able to write the program shown in Listing 5-14, once we've defin method:

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 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` name will be `can_hold`, and it will take an immutable borrow of another `Rectangle`. We can tell what the type of the parameter will be by looking at the code that `rect1.can_hold(&rect2)` passes in `&rect2`, which is an immutable borrow of `Rectangle`. This makes sense because we only need to read `rect2` (rather than write to it, which would mean we'd need a mutable borrow), and we want `main` to retain our reference to `rect1` so we can use it again after calling the `can_hold` method. The return value of `can_hold`, and the implementation will check whether the width and height of `self` are less than or equal to 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. We can take multiple parameters that we add to the signature after the `self` parameter. These additional parameters work just like regular parameters in functions.

Associated Functions

Another useful feature of `impl` blocks is that we're allowed to define functions inside them that *don't* take `self` as a parameter. These are called *associated functions* because they're associated with the struct. They're still functions, not methods, because they don't have access to the struct's fields or methods. You've already used the `String::from` associated function.

Associated functions 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 of the rectangle as a parameter and use that as both width and height, thus making it easier to create a square `Rectangle` without 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 namespace-aware, so it can be used in any module. The `square` function is part of the `Rectangle` module, which was created by `mod` in Chapter 7.

Multiple `impl` Blocks

Each struct is allowed to have multiple `impl` blocks. For example, Listing 5-16 shows some 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. We'll see a case in which multiple `impl` blocks are useful in Chapter 10 when we talk about `types` and `traits`.

Summary

Structs let you create custom types that are meaningful for your domain. By defining a struct, you keep associated pieces of data connected to each other and name each piece clearly. Methods let you specify the behavior that instances of your structs have. Functions let you namespace functionality that is particular to your struct without polluting the global namespace.

But structs aren't the only way you can create custom types: let's turn to Rust's second major feature for creating custom types—enums—in the next chapter.

Enums and Pattern Matching

In this chapter we'll look at *enumerations*, also referred to as *enums*. Enums are similar to classes in object-oriented languages, but they're more like type by enumerating its possible values. First, we'll define and use an enum to represent an IP address. Then, we'll learn how enums can encode meaning along with data. Next, we'll explore a particularly useful pattern matching construct called `match`. Finally, we'll learn how pattern matching in the `match` expression makes it easy to run different code for different variants of an enum. Finally, we'll cover how the `if let` construct is another convenient way to handle enums in your code.

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

Defining an Enum

Let's look at a situation we might want to express in code and see why enums are more appropriate than structs in this case. Say we need to work with IP addresses. There are two main standards for IP addresses: version four and version six. These are the two types of IP address that our program will come across: we can *enumerate* all possible variants of an enum to get its name.

Any IP address can be either a version four or a version six address, but not both. That property of IP addresses makes the enum data structure appropriate, because an IP address can only be one of the variants. Both version four and version six addresses are represented by the same enum type.

IP addresses, so they should be treated as the same type when the code is h
apply to any kind of IP address.

We can express this concept in code by defining an `IpAddrKind` enumeratic
possible kinds an IP address can be, `v4` and `v6`. These are known as the `va`

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

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

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
to separate the two. The reason this is useful is that now both values `IpAddrKind::V4` and
`IpAddrKind::V6` are of the same type: `IpAddrKind`. We can then, for instance, pass
that takes any `IpAddrKind`:

```
fn route(ip_type: 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
we don't have a way to store the actual IP address *data*; we only know what kind it is.
just learned about structs in Chapter 5, you might tackle this problem as an exercise.

```
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 `enum`

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

We can represent the same concept in a more concise way using just an enum inside a struct, by putting data directly into each enum variant. This new definition of `IpAddr` 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 a separate `address` field.

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 `u8` components that will have values between 0 and 255. If we wanted to store `u8` values but still express `v6` addresses as one `String` value, we wouldn't be able to do that 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 six IP addresses. However, as it turns out, wanting to store IP addresses and encode them in a standard way 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 above, but it embeds the address data inside the variants in the form of two different structures, `Ipv4Addr` and `Ipv6Addr`, 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 or structs, for example. You can even include another enum! Also, standard not much more complicated than what you might come up with.

Note that even though the standard library contains a definition for `IpAddr` use our own definition without conflict because we haven't brought the standard library 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 data 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 of data

This enum has four variants with different types:

- `Quit` has no data associated with it at all.
- `Move` includes an anonymous struct inside it.
- `Write` includes a single `String`.
- `ChangeColor` includes three `i32` values.

Defining an enum with variants like the ones in Listing 6-2 is similar to defining struct definitions, except the enum doesn't use the `struct` keyword and all variants are grouped together under the `Message` type. The following structs could hold 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 could function to take any of these kinds of messages as we could with the `Message` type 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 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.

example, we've created a variable `m` that has the value `Message::Write(Str)` and that is what `self` will be in the body of the `call` method when `m.call`. Let's look at another enum in the standard library that is very common and is extremely common in other programming languages.

The Option Enum and Its Advantages Over Null Values

In the previous section, we looked at how the `IpAddr` enum let us use Rust's more information than just the data into our program. This section explores which is another enum defined by the standard library. The `Option` type is important because it encodes the very common scenario in which a value could be something or nothing. Expressing this concept in terms of the type system means the compiler can always handle all the cases you should be handling; this functionality can be extremely common in other programming languages.

Programming language design is often thought of in terms of which features you include; 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 like C, a pointer can always be in one of two states: null or not-null.

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

I call it my billion-dollar mistake. At that time, I was designing the first computer system for references in an object-oriented language. My goal was to ensure references should be absolutely safe, with checking performed automatically. But I couldn't resist the temptation to put in a null reference, simply because I could implement. This has led to innumerable errors, vulnerabilities, and system failures. It's 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 everywhere. This kind of error.

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

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

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

The `Option<T>` enum is so useful that it's even included in the prelude; you don't need to import it into scope explicitly. In addition, so are its variants: you can use `Some` and `None` directly, without the `Option::` prefix. The `Option<T>` enum is still just a regular enum, and it has two 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 that we'll cover generics in more detail in Chapter 10. For now, all you need to know is that the `Some` variant of the `Option` enum can hold one piece of data of any type, and 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;
```

If we use `None` rather than `Some`, we need to tell Rust what type of `Option`<`T`> the compiler can't infer the type that the `Some` variant will hold by looking at.

When we have a `Some` value, we know that a value is present and the value is valid. When we have a `None` value, in some sense, it means the same thing as null. 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, Rust won't let us use an `Option<T>` value as if it were definitely a valid value. For example, consider the following code that tries 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:

```
error[E0277]: the trait bound `i8: std::ops::Add<std::option::Option` is not satisfied
-->
|
5 |     let sum = x + y;
|           ^ no implementation for `i8 + std::option::Opt-
```

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 `T`, Rust knows that we always have a valid value. We can proceed confidently without checking for null before using that value. Only when we have an `Option<i8>` (or any other type we're working with) do we have to worry about possibly not having a value, and 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 operations on it. Generally, this helps catch one of the most common issues with null: assuming something isn't null when it actually is.

Not having to worry about incorrectly assuming a not-null value helps you to write better code. In order to have a value that can possibly be null, you must explicitly declare the type of that value as `Option<T>`. Then, when you use that value, you are required to check the case when the value is null. Everywhere that a value has a type that isn't `Option<T>`, you can safely assume that the value isn't null. This was a deliberate design decision to 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>`? You can use the `as_ref` method. This method is useful in a variety of situations; you can check them out in [its documentation](#). Becoming comfortable with the various 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 handles both cases: the case when you have a `Some(T)` value, and the case when you have a `None` value. You want some code that will run only when you have a `Some(T)` value, and some other code to run if you have a `None` value. The `match` expression is a control flow construct that makes this easy.

used with enums: it will run different code depending on which variant of the code can use the data inside the matching value.

The `match` Control Flow Operator

Rust has an extremely powerful control flow operator called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern can be made up of literal values, variable names, wildcards, and many other things. This 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 each pattern is handled.

Think of a `match` expression as being like a coin-sorting machine: coins slide through a series of variously sized holes along it, and each coin falls through the first hole it encounters. In the same way, values go through each pattern in a `match`, and at the first pattern that matches, the value falls into the associated code block to be used during execution.

Because we just mentioned coins, let's use them as an example using `match` to write a function that can take an unknown United States coin and, in a similar way as with `if`, determine 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) -> u32 {
    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 patterns.

Let's break down the `match` in the `value_in_cents` function. First, we list the variants of the enum followed by an expression, which in this case is the value `coin`. This seems a lot like an `if` expression used with `if`, but there's a big difference: with `if`, the expression must evaluate to a Boolean value, but here, it can be any type. The type of `coin` in this example is the enum type we defined on line 1.

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

When the `match` expression executes, it compares the resulting value agains all the patterns in the `match` expression, in order. If a pattern matches the value, the code associated with that pattern is run. If none of the patterns match the value, execution continues to the next arm, and so on. This is like 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 last matching arm is the value that gets returned for the entire `match` expression.

Curly brackets typically aren't used if the match arm code is short, as it is in I arm just returns a value. If you want to run multiple lines of code in a match brackets. For example, the following code would print "Lucky penny!" every t called with a `Coin::Penny` but would still return the last value of the block,

```
fn value_in_cents(coin: Coin) -> u32 {
    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 th 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. In 2008, the United States minted quarters with different designs for each of th No other coins got state designs, so only quarters have this extra value. We c to our `enum` by changing the `Quarter` variant to include a `UsState` st 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`.

Let's imagine that a friend of ours is trying to collect all 50 state quarters. Wh change by coin type, we'll also call out the name of the state associated with 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 values of the variant `Coin::Quarter`. When a `Coin::Quarter` matches, the to the value of that quarter's state. Then we can use `state` in the code for tl

```
fn value_in_cents(coin: Coin) -> u32 {
    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 them match until we reach `Coin::Quarter(state)`. At that point, the binding `state` will be `UsState::Alaska`. We can then use that binding in the `println!` expression to print 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` variant of `Option<T>`; we can also handle `None` using `match` as we did with the `Quarter` enum. When comparing coins, we'll compare the variants of `Option<T>`, but the way that works remains the same.

Let's say we want to write a function that takes an `Option<i32>` and, if there is a value inside, adds one to that value. If there isn't a value inside, the function should return the `None` variant so that it can't 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 continue to the first arm of the `match` expression:

```
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` in the pattern `Some(i)` matches the `i` in the value contained in `Some`, so `i` takes the value `5`. The code in the match arm then adds 1 to the value of `i` and creates a new `Some` value with our total `6` in it.

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

```
None => None,
```

It matches! There's no value to add to, so the program stops and returns the 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 against an enum, bind a variable to the data inside, and then execute bit tricky at first, but once you get used to it, you'll wish you had it in all language favorite.

Matches Are Exhaustive

There's one other aspect of `match` we need to discuss. Consider this version 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 type error. If we try to compile this code, we'll get this error:

```
error[E0004]: non-exhaustive patterns: `None` not covered
-->
|
6 |     match x {
|         ^ pattern `None` not covered
```

Rust knows that we didn't cover every possible case and even knows which pattern we missed. 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 the `None` case, it protects us from assuming that we have a value when we're not. This is a billion-dollar mistake discussed earlier.

The `_` Placeholder

Rust also has a pattern we can use when we don't want to list all possible values. For example, if we only care about odd numbers, we can have valid values of 1 through 255. If we only care about the values 1, 3, 5, 7, 9, ..., 255, we don't have to list out 0, 2, 4, 6, 8, 10, ..., 254. Fortunately, we don't have to list them all out; we can use a special pattern `_` instead:

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

The `_` pattern will match any value. By putting it after our other arms, the `_` pattern will catch any value that wasn't covered by the previous arms.

possible cases that aren't specified before it. The `()` is just the unit value, so the `_` case. As a result, we can say that we want to do nothing for all the possibilities that don't list before the `_` placeholder.

However, the `match` expression can be a bit wordy in a situation in which we only care about one of the cases. For this situation, Rust provides `if let`.

Concise Control Flow with `if let`

The `if let` syntax lets you combine `if` and `let` into a less verbose way to match one pattern while ignoring the rest. Consider the program in Listing 6-6: it has an `Option<u8>` value but only wants to execute code if the value is `3`:

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Listing 6-6: A `match` that only cares about executing code when the value is `3`.

We want to do something with the `Some(3)` match but do nothing with any other value. To satisfy the `match` expression, we have to add `_ => ()` as a variant, which is a lot of boilerplate code to add.

Instead, we could write this in a shorter way using `if let`. The following code shows how to rewrite the `match` in Listing 6-6:

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

The syntax `if let` takes a pattern and an expression separated by an `=`. It is a shorthand for a `match`, where the expression is given to the `match` and the pattern is its first argument.

Using `if let` means you have less typing, less indentation, and less boilerplate code. However, it loses 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 worth the trade-off for losing exhaustive checking.

In other words, you can think of `if let` as syntax sugar for a `match` that runs the block of code that goes with the pattern and ignores all other values.

We can include an `else` with an `if let`. The block of code that goes with the `else` is the block of code that would go with the `_` case in the `match` expression that follows the `if let` and `else`. Recall the `Coin` enum definition in Listing 6-4, where the variants held a `UsState` value. If we wanted to count all non-quarter coins we see while counting 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 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 or enumerated values. We've shown how the standard library's `Option<T>` type system to prevent errors. When enum values have data inside them, you can extract and use those values, depending on how many cases you need to

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

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

Using Modules to Reuse and Organize

When you start writing programs in Rust, your code might live solely in the `main.rs` file. As your code grows, you'll eventually move functionality into other functions for reuse and organization. By splitting your code into smaller chunks, you make each chunk easier to understand on its own. But what happens if you have too many functions? Rust has a mechanism called modules that enables the reuse of code in an organized fashion.

In the same way that you extract lines of code into a function, you can extract larger blocks of code (like structs and enums) into different modules. A *module* is a namespace where you can define multiple items: functions, types, constants, and more. You can then choose whether those definitions are visible outside their module (public) or not (private). Here's an overview of how modules work:

- The `mod` keyword declares a new module. Code within the module appears following this declaration within curly brackets or in another file.
- By default, functions, types, constants, and modules are private. The `pub` keyword makes an item public and therefore visible outside its namespace.
- The `use` keyword brings modules, or the definitions inside modules, into your current scope so you can refer to them.

We'll look at each of these parts to see how they fit into the whole.

mod and the Filesystem

We'll start our module example by making a new project with Cargo, but instead of creating a binary crate, we'll make a library crate: a project that other people can pull into their own projects.

dependency. For example, the `rand` crate discussed in Chapter 2 is a library dependency in the guessing game project.

We'll create a skeleton of a library that provides some general networking functionality. We'll concentrate on the organization of the modules and functions, but we won't go into the function bodies. We'll call our library `communicator`. To create a library:

```
$ cargo new communicator --lib  
$ cd communicator
```

Notice that Cargo generated `src/lib.rs` instead of `src/main.rs`. Inside `src/lib.rs`:

Filename: `src/lib.rs`

```
#[cfg(test)]  
mod tests {  
    #[test]  
    fn it_works() {  
        assert_eq!(2 + 2, 4);  
    }  
}
```

Cargo creates an example test to help us get our library started. We'll look at the `mod tests` syntax in the "Using `super` to Access a Parent Module" section later; for now, leave this code at the bottom of `src/lib.rs`.

Because we don't have a `src/main.rs` file, there's nothing for Cargo to execute the `cargo run` command. Therefore, we'll use the `cargo build` command to compile our library.

We'll look at different options for organizing your library's code that will be suitable for different situations, depending on the intent of the code.

Module Definitions

For our `communicator` networking library, we'll first define a module named `network`. This is the definition of a function called `connect`. Every module definition in Rust starts with the `mod` keyword. Add this code to the beginning of the `src/lib.rs` file, above the test code:

Filename: `src/lib.rs`

```
mod network {  
    fn connect() {  
    }  
}
```

After the `mod` keyword, we put the name of the module, `network`, and then close the brackets. Everything inside this block is inside the namespace `network`. In this case, we defined a single function, `connect`. If we wanted to call this function from code outside the `network` module, we would need to specify the module and use the namespace syntax `::` like so:

We can also have multiple modules, side by side, in the same `src/lib.rs` file. For example, if we had a `client` module that has a function named `connect`, we can add it as shown below:

Filename: `src/lib.rs`

```
mod network {
    fn connect() {
    }
}

mod client {
    fn connect() {
    }
}
```

Listing 7-1: The `network` module and the `client` module defined side by side.

Now we have a `network::connect` function and a `client::connect` function with completely different functionality, and the function names do not conflict with each other because they're in different modules.

In this case, because we're building a library, the file that serves as the entry point to the library is `src/lib.rs`. However, in respect to creating modules, there's nothing stopping us from doing otherwise! We could also create modules in `src/main.rs` for a binary crate in the same way that we created modules in `src/lib.rs` for the library crate. In fact, we can put modules inside each other, which can be useful as your modules grow to keep related functionality organized together and to avoid naming conflicts. Separating functionality apart like this is called *modularization*. The way you choose to organize your code depends on how the parts of your code relate to each other. For instance, the `client` code might make more sense to users of our library if they were inside the `network` module instead, as in Listing 7-2:

Filename: `src/lib.rs`

```
mod network {
    fn connect() {
    }

    mod client {
        fn connect() {
        }
    }
}
```

Listing 7-2: Moving the `client` module inside the `network` module

In your `src/lib.rs` file, replace the existing `mod network` and `mod client` definitions with the code from Listing 7-2, which have the `client` module as an inner module of `network`. Now both `network::connect` and `network::client::connect` are both named `connect`, but they don't conflict with each other because they're in different namespaces.

In this way, modules form a hierarchy. The contents of `src/lib.rs` are at the top level, and submodules are at lower levels. Here's what the organization of our example looks like when thought of as a hierarchy:

```
communicator
└── network
    └── client
```

And here's the hierarchy corresponding to the example in Listing 7-2:

```
communicator
└── network
    └── client
```

The hierarchy shows that in Listing 7-2, `client` is a child of the `network` module sibling. More complicated projects can have many modules, and they'll need to logically group them together in order for you to keep track of them. What "logically" means in your project depends on how you and your library's users think about your project's techniques shown here to create side-by-side modules and nested modules that you would like.

Moving Modules to Other Files

Modules form a hierarchical structure, much like another structure in computer filesystems! We can use Rust's module system along with multiple files to separate code so that not everything lives in `src/lib.rs` or `src/main.rs`. For this example, let's start with:

Filename: `src/lib.rs`

```
mod client {
    fn connect() {
    }
}

mod network {
    fn connect() {
    }

    mod server {
        fn connect() {
        }
    }
}
```

Listing 7-3: Three modules, `client`, `network`, and `network::server`, all defined in `src/lib.rs`.

The file `src/lib.rs` has this module hierarchy:

```
communicator
├── client
└── network
    └── server
```

If these modules had many functions, and those functions were becoming large and difficult to scroll through this file to find the code we wanted to work with. By separating them into their own files, we can move them out of the `src/lib.rs` file and nest them inside one or more `mod` blocks, the lines of code inside the functions as well. These would be good reasons to separate the `client`, `network`, and `server` code from `src/lib.rs` and place them into their own files.

First, let's replace the `client` module code with only the declaration of the `connect` function. `src/lib.rs` looks like code shown in Listing 7-4:

Filename: `src/lib.rs`

```
mod client;

mod network {
    fn connect() {
}

mod server {
    fn connect() {
}
}
```

Listing 7-4: Extracting the contents of the `client` module but leaving the de

We're still *declaring* the `client` module here, but by replacing the block with telling Rust to look in another location for the code defined within the scope In other words, the line `mod client;` means this:

```
mod client {
    // contents of client.rs
}
```

Now we need to create the external file with that module name. Create a `cli` directory and open it. Then enter the following, which is the `connect` function module that we removed in the previous step:

Filename: `src/client.rs`

```
fn connect() {
}
```

Note that we don't need a `mod` declaration in this file because we already declared the module with `mod` in `src/lib.rs`. This file just provides the *contents* of the `client` module. If we were to declare the `mod client` here, we'd be giving the `client` module its own submodule named `client`.

Rust only knows to look in `src/lib.rs` by default. If we want to add more files to our project, we'll need to tell Rust in `src/lib.rs` to look in other files; this is why `mod client` needs to be defined in `src/client.rs`.

Now the project should compile successfully, although you'll get a few warning messages when you run `cargo build` instead of `cargo run` because we have a library crate rather than an executable.

```
$ cargo build
   Compiling communicator v0.1.0 (file:///projects/communicator)
warning: function is never used: `connect`
--> src/client.rs:1:1
  |
1 | / fn connect() {
2 | | }
  | |_ ^
  |
= note: #[warn(dead_code)] on by default

warning: function is never used: `connect`
--> src/lib.rs:4:5
  |
4 | /     fn connect() {
5 | | }
  | |____^

warning: function is never used: `connect`
--> src/lib.rs:8:9
  |
8 | /         fn connect() {
9 | | }
  | |____^
```

These warnings tell us that we have functions that are never used. Don't worry for now; we'll address them later in this chapter in the "Controlling Visibility" section. The good news is that they're just warnings; our project built successfully!

Next, let's extract the `network` module into its own file using the same pattern as before: move the body of the `network` module and add a semicolon to the declaration, like this:

Filename: `src/lib.rs`

```
mod client;

mod network;
```

Then create a new `src/network.rs` file and enter the following:

Filename: `src/network.rs`

```
fn connect() {
}

mod server {
    fn connect() {
    }
}
```

Notice that we still have a `mod` declaration within this module file; this is because we want `server` to be a submodule of `network`.

Run `cargo build` again. Success! We have one more module to extract: `server`. This is a submodule—that is, a module within a module—our current tactic of extracting modules named after that module won't work. We'll try anyway so you can see the error. Open `src/network.rs` to have `mod server;` instead of the `server` module's content.

Filename: `src/network.rs`

```
fn connect() {
}

mod server;
```

Then create a `src/server.rs` file and enter the contents of the `server` module

Filename: `src/server.rs`

```
fn connect() {
}
```

When we try to `cargo build`, we'll get the error shown in Listing 7-5:

```
$ cargo build
   Compiling communicator v0.1.0 (file:///projects/communicator)
error: cannot declare a new module at this location
--> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
  |
  note: maybe move this module `src/network.rs` to its own directory
/mod.rs`
--> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
  note: ... or maybe `use` the module `server` instead of possibly re-
--> src/network.rs:4:5
  |
4 | mod server;
  |     ^^^^^^
```

Listing 7-5: Error when trying to extract the `server` submodule into `src/server.rs`

The error says we cannot declare a new module at this location and is pointing to the `mod server;` line in `src/network.rs`. So `src/network.rs` is different than `src/lib.rs`. Let's read the note to understand why.

The note in the middle of Listing 7-5 is actually very helpful because it points us to something we haven't yet talked about doing:

```
note: maybe move this module `network` to its own directory via
`network/mod.rs`
```

Instead of continuing to follow the same file-naming pattern we used previously, the note suggests:

1. Make a new *directory* named `network`, the parent module's name.
2. Move the `src/network.rs` file into the new `network` directory and rename it to `mod.rs`.
3. Move the submodule file `src/server.rs` into the `network` directory.

Here are commands to carry out these steps:

```
$ mkdir src/network
$ mv src/network.rs src/network/mod.rs
$ mv src/server.rs src/network
```

Now when we try to run `cargo build`, compilation will work (we'll still have the same errors). The module layout still looks exactly the same as it did when we had all the code in one file.

7-3:

```

communicator
└── client
    └── network
        └── server

```

The corresponding file layout now looks like this:

```

└── src
    ├── client.rs
    ├── lib.rs
    └── network
        ├── mod.rs
        └── server.rs

```

So when we wanted to extract the `network::server` module, why did we have to move the `src/network.rs` file to the `src/network/mod.rs` file and put the code for `network` directory in `src/network/server.rs`? Why couldn't we just extract the `network::server.rs`? The reason is that Rust wouldn't be able to recognize that `server` is a submodule of `network` if the `server.rs` file was in the `src` directory. To clarify, let's consider a different example with the following module hierarchy, where `client` is in `src/lib.rs`:

```

communicator
└── client
    └── network
        └── client

```

In this example, we have three modules again: `client`, `network`, and `communicator`. Following the same steps we did earlier for extracting modules into files, we would create `src/client.rs`. Extracting the `client` module into a `src/client.rs` file is fine because that's a top-level `client` module! If we could put the code for *both* the `client` and `network::client` modules in the `src/client.rs` file, Rust wouldn't have any way to know whether the code belongs to the `client` or for `network::client`.

Therefore, in order to extract a file for the `network::client` submodule of the `client` module, we needed to create a directory for the `network` module instead of a `src/network.rs` file. If we move the `client` module into the `network` module then goes into the `src/network/mod.rs` file, and the `client` module can have its own `src/network/client.rs` file. Now the top-level `client` module clearly and unambiguously has the code that belongs to the `client` module.

Rules of Module Filesystems

Let's summarize the rules of modules with regard to files:

- If a module named `foo` has no submodules, you should put the declarations in a file named `foo.rs`.
- If a module named `foo` does have submodules, you should put the declarations in a file named `foo/mod.rs`.

These rules apply recursively, so if a module named `foo` has a submodule named `bar` and does not have submodules, you should have the following files in your `src` directory:

```

└── foo
    ├── bar.rs (contains the declarations in `foo::bar`)
    └── mod.rs (contains the declarations in `foo`, including `mod` and `bar`)

```

The modules should be declared in their parent module's file using the `mod`. Next, we'll talk about the `pub` keyword and get rid of those warnings!

Controlling Visibility with `pub`

We resolved the error messages shown in Listing 7-5 by moving the `network` code into the `src/network/mod.rs` and `src/network/server.rs` files, respectively. `cargo build` was able to build our project, but we still get warning message `client::connect`, `network::connect`, and `network::server::connect` fun

So why are we receiving these warnings? After all, we're building a library intended to be used by our *users*, not necessarily by us within our own project. That these `connect` functions go unused. The point of creating them is that another project, not our own.

To understand why this program invokes these warnings, let's try using the `extern crate` command from another project, calling it externally. To do that, we'll create a binary crate as our library crate by making a `src/main.rs` file containing this code:

Filename: `src/main.rs`

```
extern crate communicator;

fn main() {
    communicator::client::connect();
}
```

We use the `extern crate` command to bring the `communicator` library crate package now contains *two* crates. Cargo treats `src/main.rs` as the root file of a separate from the existing library crate whose root file is `src/lib.rs`. This pattern is common for executable projects: most functionality is in a library crate, and the binary crate is a separate crate. As a result, other programs can also use the library crate, and it's a nice way to share concerns.

From the point of view of a crate outside the `communicator` library looking inside, the crate being created are within a module that has the same name as the crate, `communicator`. This is called the top-level module of a crate the *root module*.

Also note that even if we're using an external crate within a submodule of our crate, the `extern crate` should go in our root module (so in `src/main.rs` or `src/lib.rs`). That way, we can refer to items from external crates as if the items are top-level modules.

Right now, our binary crate just calls our library's `connect` function from the `client` module. However, invoking `cargo build` will now give us an error after the warnings:

```
error[E0603]: module `client` is private
--> src/main.rs:4:5
  |
4 |     communicator::client::connect();
  |     ^^^^^^^^^^
```

Ah ha! This error tells us that the `client` module is private, which is the crucial part of what we've learned so far. It's also the first time we've run into the concepts of *public* and *private* in the context of Rust. In Rust, the state of all code is private: no one else is allowed to use the code. If you want to make a function public, you have to declare it as such. If you want to make a function private, you have to declare it as such. Rust will warn you that the function has gone unused.

After you specify that a function such as `client::connect` is public, not only function from your binary crate be allowed, but also the warning that the function is unused. Marking a function as public lets Rust know that the function will be used outside your program. Rust considers the theoretical external usage that's now possible "being used." Thus, when a function is marked public, Rust will not require that function to be used in your program and will stop warning that the function is unused.

Making a Function Public

To tell Rust to make a function public, we add the `pub` keyword to the start of the function definition. Let's focus on fixing the warning that indicates `client::connect` has gone unused. We'll start by marking the `client` module `client` is private error from our binary crate. Modify `src/lib.rs` to mark the `client` module public, like so:

Filename: `src/lib.rs`

```
pub mod client;  
mod network;
```

The `pub` keyword is placed right before `mod`. Let's try building again:

```
error[E0603]: function `connect` is private  
--> src/main.rs:4:5  
|  
4 |     communicator::client::connect();  
|     ^^^^^^
```

Hooray! We have a different error! Yes, different error messages are a cause of errors. This new error shows `function `connect` is private`, so let's edit `src/client.rs` to mark the `client::connect` function public too:

Filename: `src/client.rs`

```
pub fn connect() {  
}
```

Now run `cargo build` again:

```
warning: function is never used: `connect`  
--> src/network/mod.rs:1:1  
|  
1 | / fn connect() {  
2 | | }  
| |_-^  
|= note: #[warn(dead_code)] on by default  
  
warning: function is never used: `connect`  
--> src/network/server.rs:1:1  
|  
1 | / fn connect() {  
2 | | }  
| |_-^
```

The code compiled, and the warning that `client::connect` is not being used is gone!

Unused code warnings don't always indicate that an item in your code needs to be removed. You might be using it internally or you *didn't* want these functions to be part of your public API, unused code warnings are useful for catching those cases.

you to code you no longer need that you can safely delete. They could also be useful if you had just accidentally removed all places within your library where this function was used.

But in this case, we *do* want the other two functions to be part of our crate's public API. We can do this by marking them as `pub` as well to get rid of the remaining warnings. Modify `src/network/mod.rs` to look like the following:

Filename: `src/network/mod.rs`

```
pub fn connect() {  
}  
  
mod server;
```

Then compile the code:

```
warning: function is never used: `connect`  
--> src/network/mod.rs:1:1  
|  
1 | / pub fn connect() {  
2 | | }  
| |_-^  
|= note: #[warn(dead_code)] on by default  
  
warning: function is never used: `connect`  
--> src/network/server.rs:1:1  
|  
1 | / fn connect() {  
2 | | }  
| |_-^
```

Hmm, we're still getting an unused function warning, even though `network` is `pub`. The reason is that the function is public within the module, but the `network` module itself is not public. We're working from the interior of the library, so the `client::connect` function resides in `client`, which is not public. We need to change `server` to be `pub` too, like so:

Filename: `src/lib.rs`

```
pub mod client;  
  
pub mod network;
```

Now when we compile, that warning is gone:

```
warning: function is never used: `connect`  
--> src/network/server.rs:1:1  
|  
1 | / fn connect() {  
2 | | }  
| |_-^  
|= note: #[warn(dead_code)] on by default
```

Only one warning is left—try to fix this one on your own!

Privacy Rules

Overall, these are the rules for item visibility:

- If an item is public, it can be accessed through any of its parent modules.
- If an item is private, it can be accessed only by its immediate parent module or its parent's child modules.

Privacy Examples

Let's look at a few more privacy examples to get some practice. Create a new file named `src/lib.rs` and enter the code in Listing 7-6 into your new project's `src/lib.rs`:

Filename: `src/lib.rs`

```
mod outermost {  
    pub fn middle_function() {}  
  
    fn middle_secret_function() {}  
  
    mod inside {  
        pub fn inner_function() {}  
  
        fn secret_function() {}  
    }  
}  
  
fn try_me() {  
    outermost::middle_function();  
    outermost::middle_secret_function();  
    outermost::inside::inner_function();  
    outermost::inside::secret_function();  
}
```

Listing 7-6: Examples of private and public functions, some of which are inaccessible from the root module

Before you try to compile this code, make a guess about which lines in the code will cause errors. Then, try compiling the code to see whether you were right—and read the error messages!

Looking at the Errors

The `try_me` function is in the root module of our project. The module name is `outermost`, but the second privacy rule states that the `try_me` function is allowed to access any public functions in any module because `outermost` is in the current (root) module, as is `try_me`.

The call to `outermost::middle_function` will work because `middle_function` is public. The call to `outermost::middle_secret_function` will not work because `middle_secret_function` is private, the second rule applies. The root module is neither the parent module of `middle_secret_function` (`outermost` is), nor is it a child module of `middle_secret_function`.

The module named `inside` is private and has no child modules, so it can be accessed only from its parent module `outermost`. That means the `try_me` function is not allowed to access either `outermost::inside::inner_function` OR `outermost::inside::secret_function`.

Fixing the Errors

Here are some suggestions for changing the code in an attempt to fix the errors:

whether it will fix the errors before you try each one. Then compile the code you're right, using the privacy rules to understand why. Feel free to design more of them out!

- What if the `inside` module were public?
- What if `outermost` were public and `inside` were private?
- What if, in the body of `inner_function`, you called `::outermost::mid`? (The two colons at the beginning mean that we want to refer to the module's root module.)

Next, let's talk about bringing items into scope with the `use` keyword.

Referring to Names in Different Modules

We've covered how to call functions defined within a module using the module's name in its path, as in the call to the `nested_modules` function shown here in Listing 7-7:

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

fn main() {
    a::series::of::nested_modules();
}
```

Listing 7-7: Calling a function by fully specifying its enclosing module's path

As you can see, referring to the fully qualified name can get quite lengthy. Fortunately, Rust has a keyword to make these calls more concise.

Bringing Names into Scope with the `use` Keyword

Rust's `use` keyword shortens lengthy function calls by bringing the modules or crates that we want to call into scope. Here's an example of bringing the `a::series::of` module into scope for the current crate's root scope:

Filename: src/main.rs

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of;

fn main() {
    of::nested_modules();
}
```

The line `use a::series::of;` means that rather than using the full `a::series` module, we want to refer to the `of` module, we can use `of`.

The `use` keyword brings only what we've specified into scope: it does not bring all of `a` into scope. That's why we still have to use `of::nested_modules` when we want to call the `nested_modules` function.

We could have chosen to bring the function into scope by instead specifying the entire module as follows:

```
pub mod a {
    pub mod series {
        pub mod of {
            pub fn nested_modules() {}
        }
    }
}

use a::series::of::nested_modules;

fn main() {
    nested_modules();
}
```

Doing so allows us to exclude all the modules and reference the function directly as `nested_modules()`.

Because enums also form a sort of namespace like modules, we can bring all variants of an enum into scope with `use` as well. For any kind of `use` statement, if you're bringing multiple items from a single namespace into scope, you can list them using curly brackets and commas instead of colons, so:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::{Red, Yellow};

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = TrafficLight::Green;
}
```

We're still specifying the `TrafficLight` namespace for the `Green` variant because we're using the colon operator in the `use` statement.

Nested groups in `use` declarations

If you have a complex module tree with many different submodules and you want to bring specific items from each one, it might be useful to group all the imports in the same `use` declaration to keep your code clean and avoid repeating the base modules' name.

The `use` declaration supports nesting to help you in those cases, both with top-level and nested `use` declarations. For example this snippet imports `bar`, `baz`, `Foo`, and `Quux`, all the items in `baz` and `bar` respectively.

```
use foo::{
    bar::{self, Foo},
    baz::{*, quux::Bar},
};
```

Bringing All Names into Scope with a Glob

To bring all the items in a namespace into scope at once, we can use the `*:` the *glob operator*. This example brings all the variants of an enum into scope each specifically:

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}

use TrafficLight::*;

fn main() {
    let red = Red;
    let yellow = Yellow;
    let green = Green;
}
```

The `*` will bring into scope all the visible items in the `TrafficLight` namespace sparingly: they are convenient, but a glob might also pull in more items cause naming conflicts.

Using `super` to Access a Parent Module

As you saw at the beginning of this chapter, when you create a library crate, module for you. Let's go into more detail about that now. In your `communicator` `src/lib.rs`:

Filename: `src/lib.rs`

```
pub mod client;

pub mod network;

#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Chapter 11 explains more about testing, but parts of this example should make a module named `tests` that lives next to our other modules and contains one `it_works`. Even though there are special annotations, the `tests` module is part of our module hierarchy looks like this:

```
communicator
├── client
├── network
│   └── client
└── tests
```

Tests are for exercising the code within our library, so let's try to call our `client` from this `it_works` function, even though we won't be checking any function yet: it won't work yet:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        client::connect();
    }
}
```

Run the tests by invoking the `cargo test` command:

```
$ cargo test
    Compiling communicator v0.1.0 (file:///projects/communicator)
error[E0433]: failed to resolve. Use of undeclared type or module
--> src/lib.rs:9:9
   |
9 |         client::connect();
   |         ^^^^^^ Use of undeclared type or module `client`
```

The compilation failed, but why? We don't need to place `communicator::` in we did in `src/main.rs`, because we are definitely within the `communicator` lib reason is that paths are always relative to the current module, which here is exception is in a `use` statement, where paths are relative to the crate root b module needs the `client` module in its scope!

So how do we get back up one module in the module hierarchy to call the `c` function in the `tests` module? In the `tests` module, we can either use `leaf` know that we want to start from the root and list the whole path, like this:

```
::client::connect();
```

Or, we can use `super` to move up one module in the hierarchy from our cur

```
super::client::connect();
```

These two options don't look that different in this example, but if you're deep hierarchy, starting from the root every time would make your code lengthy. `super` to get from the current module to sibling modules is a good shortcut the path from the root in many places in your code and then rearrange your subtree to another place, you'll end up needing to update the path in several tedious.

It would also be annoying to have to type `super::` in each test, but you've a that solution: `use !` The `super::` functionality changes the path you give to the parent module instead of to the root module.

For these reasons, in the `tests` module especially, `use super::something` solution. So now our test looks like this:

Filename: `src/lib.rs`

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

    #[test]
    fn it_works() {
        client::connect();
    }
}
```

When we run `cargo test` again, the test will pass, and the first part of the terminal output will look like the following:

```
$ cargo test
   Compiling communicator v0.1.0 (file:///projects/communicator)
     Running target/debug/communicator-92007ddb5330fa5a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Summary

Now you know some new techniques for organizing your code! Use these techniques to keep related functionality together, keep files from becoming too long, and present your library users.

Next, we'll look at some collection data structures in the standard library that will help you write nice, neat code.

Common Collections

Rust's standard library includes a number of very useful data structures called collections. Unlike primitive data types which represent one specific value, but collections can contain multiple values. Unlike arrays and tuples, the data these collections point to is stored on the heap. The amount of data does not need to be known at compile time and can grow or shrink over time. Each kind of collection has different capabilities and costs, and choosing the right one for your current situation is a skill you'll develop over time. In this chapter, we'll look at 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 earlier 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 specialized 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 [standard library documentation](#).

We'll discuss how to create and update vectors, strings, and hash maps, as well as some special cases.

Storing Lists of Values with Vectors

The first collection type we'll look at is `Vec<T>`, also known as a `vector`. Vectors are lists that store more than one value in a single data structure that puts all the values next to each other. Vectors can only store values of the same type. They are useful when you have many values of the same type together, 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 can call the `Vec::new` function, as shown

```
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, Rust doesn't know what kind of elements we intend to store. This is an implementation detail of vectors; we'll cover how to use generics with your own types later. For now, know that the `Vec<T>` type provided by the standard library can hold a specific type of value. The type is specified within angle brackets, telling Rust that the `Vec<T>` in `v` will hold elements of the `i32` type.

In more realistic code, Rust can often infer the type of value you want to store in a vector, so you rarely need to do this type annotation. It's more common to create a vector with initial values, and Rust provides the `vec!` macro for convenience. The macro creates a new vector that holds the values you give it. Listing 8-2 creates a new `Vec<i32>` that contains the values `1`, `2`, and `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>`. 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. 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 a sequence of literals, 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 will be cleaned up. This may seem like a straightforward point but can get a bit more interesting when you start to introduce references to the elements of the vector. Let's take a look.

Reading Elements of Vectors

Now that you know how to create, update, and destroy vectors, knowing how to read from them is a good next step. There are two ways to reference a value stored in a vector: one way uses indexing syntax, and the other way we've annotated the types of the values that are returned from these functions.

Listing 8-5 shows the method of accessing a value in a vector with indexing syntax.

```
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2];
```

Listing 8-5: Using indexing syntax to access an item in a vector

Listing 8-6 shows the method of accessing a value in a vector, with the `get` method.

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

match v.get(v_index) {
    Some(_) => { println!("Reachable element at index: {}", v_index); }
    None => { println!("Unreachable element at index: {}", v_index); }
}
```

Listing 8-6: Using 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 in the vector, indexed by number, starting at zero. Second, the two ways to get the third element are `v[2]`, which gives us a reference, or by using the `get` method with the index as an argument, which gives us an `Option<&T>`.

Rust has two ways to reference an element so you can choose how the program behaves if you try to use an index value that the vector doesn't have an element for. As an example, consider what the program will do if it has a vector that holds five elements and then tries to attempt to access the element at index 100, as shown in Listing 8-7:

```
let v = vec![1, 2, 3, 4, 5];
let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

Listing 8-7: 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 nonexistent element. This method is best used when you want your program 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 panicking. You would use this method if accessing an element beyond the range happens occasionally under normal circumstances. Your code will then have either `Some(&element)` or `None`, as discussed in Chapter 6. For example, the `get` method could come from a person entering a number. If they accidentally enter a number that's out of range, the program gets a `None` value, you could tell the user how many items are in the vector and ask them to enter another number. 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 rules (covered in Chapter 4) to ensure this reference and any other references to the same vector remain valid. Recall the rule that states you can't have mutable and immutable references to the same scope. That rule applies in Listing 8-8, where we hold an immutable reference to the first element in a vector and try to add an element to the end, which won't work:

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
```

Listing 8-8: Attempting to add an element to a vector while holding a reference to its first element
Compiling this code will result in this error:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
-->
|
4 |     let first = &v[0];
|             - immutable borrow occurs here
5 |
6 |     v.push(6);
|         ^ mutable borrow occurs here
7 |
8 | }
```

The code in Listing 8-8 might look like it should work: why should a reference to the first element care about what changes at the end of the vector? This error is due to the way memory works: adding a new element onto the end of the vector might require allocating new memory for all the elements to the new space, if there isn't enough room to put all the elements in the same place where the vector currently is. In that case, the reference to the first element would point to deallocated memory. The borrowing rules prevent programs from ending up in such situations.

Note: For more on the implementation details of the `Vec<T>` type, see “Implementation Details” in the Nomicon at <https://doc.rust-lang.org/stable/nomicon/vec.html>.

Iterating over the Values in a Vector

If we want to access each element in a vector in turn, we can iterate through the vector rather than use indexes to access one at a time. Listing 8-9 shows how to use immutable references to each element in a vector of `i32` values and print their values.

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

Listing 8-9: Printing each element in a vector by iterating over the elements ↴

We can also iterate over mutable references to each element in a mutable vector. If we change to all the elements. The `for` loop in Listing 8-10 will add `50` to each element.

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

Listing 8-10: Iterating over mutable references to elements in a vector ↴

To change the value that the mutable reference refers to, we have to use the `*` operator to get to the value in `i` before we can use the `+=` operator. We'll talk more about mutable references in Chapter 15.

Using an Enum to Store Multiple Types

At the beginning of this chapter, we said that vectors can only store values of one type. This can be inconvenient; there are definitely use cases for needing to store different types. Fortunately, the variants of an enum are defined under the same `enum` name, so when we need to store elements of a different type in a vector, we can define a new variant.

For example, say we want to get values from a row in a spreadsheet in which the row contains integers, some floating-point numbers, and some strings. We can define an enum whose variants will hold the different value types, and then all the enum variants will have the same type: that of the enum. Then we can create a vector that holds that type, which ultimately holds different types. We've demonstrated this in Listing 8-11:

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

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

Listing 8-11: 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. A secondary advantage of using an enum is that it makes it explicit about what types are allowed in this vector. If Rust allowed a vector to contain any type, there would be a chance that one or more of the types would cause errors with the rest of the code. Using an enum plus a `match` expression means that at compile time that every possible case is handled, as discussed in Chapter 10.

When you're writing a program, if you don't know the exhaustive set of types that you want to runtime to store in a vector, the enum technique won't work. Instead, you can use a trait.

which we'll cover in Chapter 17.

Now that we've discussed some of the most common ways to use vectors, let's look at strings. The documentation for all the many useful methods defined on `Vec<T>` by the standard library is available online. For example, in addition to `push`, a `pop` method removes and returns the last element from the vector. There are also methods for reversing the vector or finding the index of 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. One reason you might commonly get stuck on strings due to a combination of three reasons: Rust's ownership system, exposing possible errors, strings being a more complicated data structure than other collections, and UTF-8. These factors combine in a way that can seem counterintuitive compared to coming from other programming languages.

It's useful to discuss strings in the context of collections because strings are a collection of bytes, plus some methods to provide useful functionality when interpreted as text. In this section, we'll talk about the operations on `String` type that the standard library provides, such as creating, updating, and reading. We'll also discuss the ways in which strings differ from the other collections, namely how indexing into a `String` is different from indexing into a `Vec`. Finally, we'll look at 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, which is the string slice `&str` that is usually seen in its borrowed form. You might have noticed that 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 binary output of the `println!` macro, therefore string slices.

The `String` type, which is provided by Rust's standard library rather than the `std::str` module, is a growable, mutable, owned, UTF-8 encoded string type. When I say "strings" in Rust, they usually mean the `String` and the string slice `&str` types. Although this section is largely about `String`, both types are used here. The `String` type is provided by the 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 `CString`, `String`, and `CStr`. Library crates can provide even more options for storing strings. Those names all end in `String` or `Str`? They refer to owned and borrowed `String` and `Str` types you've seen previously. These string types can store different encodings or be represented in memory in a different way, for example. We'll look at other string types in this chapter; see their API documentation for more about each type and when each is appropriate.

Creating a New String

Many of the same operations available with `Vec<T>` are available with `String`. For example, 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 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

This code creates a string containing `initial contents`.

We can also use the function `String::from` to create a `String` from a string literal. 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

Because strings are used for so many things, we can use many different general functions to work with them, providing us with a lot of options. Some of them can seem redundant, but they serve different purposes. In this case, `String::from` and `to_string` do the same thing, so which you use depends on your personal style.

Remember that strings are UTF-8 encoded, so we can include any properly encoded character. Listing 8-14 shows how to store greetings in multiple languages:

```
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("নমস্তা");
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 vector. You can push more data into it. In addition, you can conveniently use the `+` operator 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 to the end of the current string.

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 `str` slice, so we don't necessarily want to take ownership of the parameter. For example, the following code shows that it would be unfortunate if we weren't able to use `s2` after appending 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 later. However, this code works as we'd expect!

The `push` method takes a single character as a parameter and adds it to the end of the string. The following code shows code that 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 of this code, `s` will contain `lol`.

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

Often, you'll want to combine two existing strings. One way is to use the `+` operator. 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`

The string `s3` will contain `Hello, world!` as a result of this code. The reason for this is that the `+` operator is implemented as a `String` method. After the addition, the variable `s1` no longer contains `Hello,` because the `String` type is immutable. The reason we used a reference to `s2` has to do with how the `+` operator is implemented. The `+` operator uses the `add` method, which is a `String` method. This method that gets called when we use the `+` operator. The `+` operator uses the `add` method, which is a `String` method. The signature looks something like this:

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

This isn't the exact signature that's in the standard library: in the standard library, the `add` method is generic. Here, we're looking at the signature of `add` with concrete types. In fact, the `String` type is itself a generic type, which is what happens when we call this method with `String` as the type. This signature gives us the clues we need to understand how the `+` operator works.

First, `s2` has an `&`, meaning that we're adding a *reference* of the second string because of the `s` parameter in the `add` function: we can only add a `&str` to two `String` values together. But wait—the type of `&s2` is `&String`, not `&str`. 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 argument into a `&str`. When we call the `add` method, Rust uses a *deref coercion* to turn `&s2` into `&s2[..]`. We'll discuss deref coercion in more depth in Chapter 15, but take ownership of the `s` parameter, `s2` will still be a valid `String` after this.

Second, we can see in the signature that `add` takes ownership of `self`, because it has an `&`. This means `s1` in Listing 8-18 will be moved into the `add` call and owned by `s2` after that. So although `let s3 = s1 + &s2;` looks like it will copy both strings, 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 the implementation is more efficient than copying.

If we need to concatenate multiple strings, the behavior of the `+` operator gets a bit more complex:

```
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 easy to make mistakes when concatenating strings. For more complicated string combining, we can 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 in the same way as the `println!` macro, but instead of printing the output to the screen, it returns a `String` with the desired output. Using `format!` is much easier to read and doesn't take ownership of the strings it concatenates.

Indexing into Strings

In many other programming languages, accessing individual characters in a string by index is a valid and common operation. However, if you try to access a character in 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:

```
error[E0277]: the trait bound `std::string::String: std::ops::Index`
satisfied
-->
|
3 |     let h = s1[0];
|           ^^^^^^ the type `std::string::String` cannot be indexed
|
= help: the trait `std::ops::Index<{integer}>` is not implemented for
`std::string::String`
```

The error and the note tell the story: Rust strings don't support indexing. But 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 strings from Listing 8-14. First, this one:

```
let len = String::from("Hola").len();
```

In this case, `len` will be 4, which means the vector storing the string "Hola" in memory contains four bytes. (Each of these letters takes 1 byte when encoded in UTF-8. But what about the following line? It begins with the capital Cyrillic letter Ze, not the Arabic number 3.)

```
let len = String::from("Здравствуйте").len();
```

Asked how long the string is, you might say 12. However, Rust's answer is 24. That's because it takes 24 bytes to encode "Здравствуйте" in UTF-8, because each Unicode scalar value requires 4 bytes of storage. Therefore, an index into the string's bytes will not always correlate to the same character value. To demonstrate, consider this invalid Rust code:

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

What should the value of `answer` be? Should it be `3`, the first letter? When encoding the string "Здравствуйте" in UTF-8, the first byte is `208` and the second is `151`, so `answer` should in fact be `2`. That's because `3` is a valid character on its own. Returning `208` is likely not what a user would want. If they wanted the first letter of this string, however, that's the only data that Rust has at byte index `0`. If they don't want the byte value returned, even if the string contains only Latin letters, there's no way to tell Rust that. If they wrote valid code that returned the byte value, it would return `104`, not `h`. To prevent such bugs, Rust's string type has a method that returns the first character as a `char` instead of a `u8`. This makes it clear that the string starts with the letter "Z". This is a good example of how Rust's type system can prevent unexpected bugs that might not be discovered immediately or even 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 a string from a perspective: as bytes, scalar values, and grapheme clusters (the closest thing to characters).

If we look at the Hindi word "नमस्ते" written in the Devanagari script, it is stored in memory as a sequence of bytes that looks like this:

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

That's 18 bytes and is how computers ultimately store this data. If we look at the string "नमस्ते" in memory, we see something like this:

values, which are what Rust's `char` type is, those bytes look like this:

```
['\u{041e}', '\u{041f}', '\u{041d}', '\u{0417}', '\u{0412}', '\u{0410}']
```

There are six `char` values here, but the fourth and sixth are not letters: they make sense on their own. Finally, if we look at them as grapheme clusters, we 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 your program can choose the interpretation it needs, no matter what human language it is.

A final reason Rust doesn't allow us to index into a `String` to get a character is that string operations are expected to always take constant time ($O(1)$). But it isn't possible to maintain this performance with a `String`, because Rust would have to walk through the characters 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 value of an indexing operation should be: a byte value, a character, a grapheme cluster, or something else entirely. Therefore, Rust asks you to be more specific if you really need to use indices. To be more specific in your indexing and indicate that you want a string slice, use `[]` with a single number, you can use `[]` with a range to create a string slice of 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 saw that the length of each of these characters was 2 bytes, which means `s` will be `Зд`.

What would happen if we used `&hello[0..1]`? The answer: Rust would panic with a `byte index 1 is not a char boundary; it is outside the range [0..2)` error. This is because the string starts at index 0, so index 1 is not a valid character boundary.

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is outside the range [0..2)' of `Здравствуйте`, src/libcore/str/mod.rs:2188:4
```

You should use ranges to create string slices with caution, because doing so can lead to undefined behavior.

Methods for Iterating Over Strings

Fortunately, you can access elements in a string in other ways.

If you need to perform operations on individual Unicode scalar values, the best way is to use the `chars` method. Calling `chars` on `"नमस्ते"` separates out and returns each character, and you can iterate over the result in order to access each element:

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

This code will print the following:

ন
ম
স
ু
ত
০

The `bytes` method returns each raw byte, which might be appropriate for you.

```
for b in "নমস্তে".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 multiple bytes.

Getting grapheme clusters from strings is complex, so this functionality is not part of 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 must decide how to present this complexity to the programmer. Rust has chosen to handle strings in a simple way: the default behavior for all Rust programs, which means you don't have to put more thought into handling UTF-8 data upfront. This trade-off eliminates some of the complexity of strings than is apparent in other programming languages, but it means you have to handle errors involving non-ASCII characters later in your development process.

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 keys of type `K` to values of type `V`. It does this via a *hashing function*, which places these keys and values into memory. Many programming languages support this structure, but they often use a different name, such as `hash_map`, `map`, `object`, `hashtable`, or `array`, just to name a few.

Hash maps are useful when you want to look up data not by using an index, but by using a key that can be of any type. For example, in a game, you could store each team's score in a hash map in which each key is a team's name and the value is the 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 good functions are defined on `HashMap<K, V>` by the standard library. As always, check the documentation for more information.

Creating a New Hash Map

You can create an empty hash map with `new` and add elements with `insert` keeping track of the scores of two teams whose names are Blue and Yellow, 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 our three common collections, this one is the least often used, so it's not included in the prelude. Hash maps also have less 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 integer values of type `i32`. Like vectors, hash maps are homogeneous: all of the keys are the same type, and all of the values must have the same type.

Another way of constructing a hash map is by using the `collect` method or where each tuple consists of a key and its value. The `collect` method gathers collection types, including `HashMap`. For example, if we had the team names in two separate vectors, we could use the `zip` method to create a vector of tuples paired with 10, and so forth. Then we could use the `collect` method to turn that 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 scores: HashMap<_, _> = teams.iter().zip(initial_scores.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 create different data structures and Rust doesn't know which you want unless you provide parameters for the key and value types, however, we use underscores, and let the compiler figure out that the hash map contains based on the types of the data in the vectors.

Hash Maps and Ownership

For types that implement the `Copy` trait, like `i32`, the values are copied into owned values like `String`, the values will be moved and the hash map will keep 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
// see what compiler error you get!

```

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

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

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

Accessing Values in a Hash Map

We can get a value out of the hash map by providing its key to the `get` method. 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 value will be wrapped in `Some(10)`. The result is wrapped in `Some` because `get` returns an `Option`. If there is no value for that key in the hash map, `get` will return `None`. The program will need to handle this case using 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 did 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 keys and values 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 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 add the new value if the key doesn't already have a value. Or you could replace 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 again, the value associated with that key will be replaced. Even though the code in Listing 8-24 is run twice, the hash map will only contain one key/value pair because we're inserting the same 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 by the new value.

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 new value. Hash maps have a special API for this called `entry` that takes the key you want to insert. The return value of the `entry` function is an enum called `Entry` that represents either an existing value or a new value that might not exist. Let's say we want to check whether the key for the color `Yellow` has a value associated with it. If it doesn't, we want to insert the value `50`, and the same value for the color `Blue`. 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 exist

The `or_insert` method on `Entry` is defined to return a mutable reference to the corresponding `Entry` if that key exists, and if not, inserts the parameter provided and returns a mutable reference to the new value. This technique is called `atomicwrites`.

writing the logic ourselves and, in addition, plays more nicely with the borrowing rules.

Running the code in Listing 8-25 will print `{"Yellow": 50, "Blue": 10}`. The first call to `entry` inserts the key for the Yellow team with the value `50` because the Yellow team was not in the map yet. The second call to `entry` will not change the hash map because the key `"Blue"` 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. 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 for each word we've seen. If it's the first time we've seen a word, we'll insert it with a value of `1`.

```
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 word counts.

This code will print `{"world": 2, "hello": 1, "wonderful": 1}`. The `or_insert` method returns a mutable reference (`&mut v`) to the value for this key. Here we store the mutable reference in the `count` variable, so in order to assign to that value, we must use the asterisk (`*`). The mutable reference goes out of scope at the end of the block, so these changes are safe and allowed by the borrowing rules.

Hashing Functions

By default, `HashMap` uses a cryptographically secure hashing function that can't be broken by Denial of Service (DoS) attacks. This is not the fastest hashing algorithm available, but the performance hit for better security that comes with the drop in performance is worth it. If you find that the default hash function is too slow for your purposes, you can switch to a faster one by specifying a different *hasher*. A hasher is a type that implements the `BuildHasher` trait. Learn about traits and how to implement them in Chapter 10. You don't necessarily have to implement the trait yourself; [crates.io](#) has libraries shared by other Rust users that contain hashers implementing many common hashing algorithms.

Summary

Vectors, strings, and hash maps will provide a large amount of functionality in your programs when you need to store, access, and modify data. Here are some exercises you can work through to get equipped to solve:

- Given a list of integers, use a vector and return the mean (the average of all the numbers), median (the middle number when the numbers are sorted), and mode (the value that occurs most frequently).

map will be helpful here) of the list.

- Convert strings to pig latin. The first consonant of each word is moved and “ay” is added, so “first” becomes “irst-fay.” Words that start with a vowel move to the end instead (“apple” becomes “apple-hay”). Keep in mind the details of encoding!
- Using a hash map and vectors, create a text interface to allow a user to add people to a department in a company. For example, “Add Sally to Engineering” Then let the user retrieve a list of all people in a department or all people in a 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, in the next chapter we’ll discuss error handling. We’ll do that next!

Error Handling

Rust’s commitment to reliability extends to error handling. Errors are a fact of life, and Rust has a number of features for handling situations in which something goes wrong. In most cases, Rust requires you to acknowledge the possibility of an error and take appropriate action to handle it. This requirement makes your program more robust by encouraging you to discover errors and handle them appropriately before you’ve deployed your program.

Rust groups errors into two major categories: *recoverable* and *unrecoverable*. For example, if you’re trying to read a file and the file does not exist, that’s a recoverable error, such as a file not found error, it’s reasonable to report the error and retry the operation. Unrecoverable errors are always symptoms of bugs in your code, such as attempting to access memory at a location beyond the end of an array.

Most languages don’t distinguish between these two kinds of errors and handle them in the same way, using mechanisms such as exceptions. Rust doesn’t have exceptions. Instead, it uses the `Result<T, E>` type for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. This chapter covers calling `panic!` and other ways of handling errors, as well as about returning `Result<T, E>` values. Additionally, we’ll explore considerations for 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. Rust has the `panic!` macro. When the `panic!` macro executes, your program prints an error message, unwind and clean up the stack, and then quit. This most commonly happens when some kind of error occurs and it’s not clear to the programmer how to handle it.

Unwinding the Stack or Aborting in Response to a Panic

By default, when a panic occurs, the program starts *unwinding*, which means it unwinds the stack and cleans up the data from each function it encounters. But unwinding and cleanup is a lot of work. The alternative is to immediately *abort*, which means the program exits without cleaning up. Memory that the program was using will then need to be freed by the operating system. If in your project you need to make the resulting binary smaller, you can switch from unwinding to aborting upon a panic by adding the appropriate `[profile]` sections in your `Cargo.toml` file. For example:

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 0.25 secs
    Running `target/debug/panic`
thread 'main' panicked at 'crash and burn', src/main.rs:2:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

The call to `panic!` causes the error message contained in the last three line our panic message and the place in our source code where the panic occurs indicates that it's the second line, fourth 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 call. In other cases, the `panic!` call might be in code that our code calls, and number reported by the error message will be someone else's code where the called, not the line of our code that eventually led to the `panic!` call. We can the functions the `panic!` call came from to figure out the part of our code the problem. We'll discuss what a backtrace is 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 of a bug in our code instead of from our code calling the macro directly. Listi that attempts to access an element by index in a vector:

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, whi

Here, we're attempting to access the hundredth element of our vector (which indexing starts at zero), but it has only three elements. In this situation, Rust supposed to return an element, but if you pass an invalid index, there's no e return here that would be correct.

Other languages, like C, will attempt to give you exactly what you asked for i though it isn't what you want: you'll get whatever is at the location in memor to that element in the vector, even though the memory doesn't belong to the *buffer overread* and can lead to security vulnerabilities if an attacker is able to

in such a way as to read data they shouldn't be allowed to that is stored after

To protect your program from this sort of vulnerability, if you try to read 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 0.27 secs
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', /checkout/src/liballoc/vec.rs:1555:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

This error points at a file we didn't write, `vec.rs`. That's the implementation of the standard library. The code that gets run when we use `[]` on our vector `v` is in `vec.rs`, and a `panic!` is actually happening.

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 frames showing where each function was called to get to this point. Backtraces in Rust work as they do in other languages: to reading the backtrace is to start from the top and read until you see files you recognize, which is usually the spot where the problem originated. The lines above the lines mentioning your code are code that was called by your code; the lines below are code that called your code. These lines are usually your 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 what you'll see:

```
$ RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but tl
/checkout/src/liballoc/vec.rs:1555:10
stack backtrace:
  0: std::sys::imp::backtrace::tracing::imp:: unwind_backtrace
      at /checkout/src/libstd/sys/unix/backtrace/tracing/gc
  1: std::sys_common::backtrace::_print
      at /checkout/src/libstd/sys_common/backtrace.rs:71
  2: std::panicking::default_hook::{closure}
      at /checkout/src/libstd/sys_common/backtrace.rs:60
      at /checkout/src/libstd/panicking.rs:381
  3: std::panicking::default_hook
      at /checkout/src/libstd/panicking.rs:397
  4: std::panicking::rust_panic_with_hook
      at /checkout/src/libstd/panicking.rs:611
  5: std::panicking::begin_panic
      at /checkout/src/libstd/panicking.rs:572
  6: std::panicking::begin_panic_fmt
      at /checkout/src/libstd/panicking.rs:522
  7: rust_begin_unwind
      at /checkout/src/libstd/panicking.rs:498
  8: core::panicking::panic_fmt
      at /checkout/src/libcore/panicking.rs:71
  9: core::panicking::panic_bounds_check
      at /checkout/src/libcore/panicking.rs:58
 10: <alloc::vec::Vec<T> as core::ops::index::Index<usize>>::inde
      at /checkout/src/liballoc/vec.rs:1555
 11: panic::main
      at src/main.rs:4
 12: __rust_maybe_catch_panic
      at /checkout/src/libpanic_unwind/lib.rs:99
 13: std::rt::lang_start
      at /checkout/src/libstd/panicking.rs:459
      at /checkout/src/libstd/panic.rs:361
      at /checkout/src/libstd/rt.rs:61
 14: main
 15: __libc_start_main
 16: <unknown>
```

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

That's a lot of output! The exact output you see might be different depending on your system and Rust version. In order to get backtraces with this information, `RUST_BACKTRACE=1` must be enabled. Debug symbols are enabled by default when using `cargo build --release`, as we have here.

In the output in Listing 9-2, line 11 of the backtrace points to the line in our program where the panic occurred: line 4 of `src/main.rs`. If we don't want our program to panic, the location of the panic is where we should start investigating. In this case, we deliberately wrote code that would panic in order to demonstrate how to use `unwrap`. To fix the panic, we could change the value of `vec[99]` to something other than `99`. Another way to fix the panic is to not request an element at index 99 from a vector that only has three elements. If you're writing code that might panic in the future, you'll need to figure out what action the code should take when it panics and what the code should do instead.

We'll come back to `panic!` and when we should and should not use `panic!` in the "To `panic!` or Not to `panic!`?" section later in this chapter. For now, let's learn 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. So instead of crashing, the 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 print an error message and continue executing the rest of your program instead of terminating the process.

Recall from “[Handling Potential Failure with the Result Type](#)” in Chapter 2 that the `Result` type 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 later. What you need to know right now is that `T` represents the type of the value that was successfully produced by the function, and `E` represents the type of the error that occurred during the failure case within the `Err` variant. Because `Result` has these generic type parameters, we can use it to represent the return value of a function in situations where the successful value and error value we want to return may have different types.

Let's call a function that returns a `Result` value because the function could fail: opening 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 documentation, or we could ask the compiler! If we give `f` a type annotation that doesn't match the return type of the function and then try to compile the code, the compiler will tell us what the type of `f` is. Let's try this with the `File::open` function: its return type is `Result<File, Error>`. The return type of `File::open` isn't of type `u32`, so let's change the `let f` statement to give it the type `Result<File, Error>`:

```
let f: Result<File, Error> = File::open("hello.txt");
```

Attempting to compile now gives us the following output:

```
error[E0308]: mismatched types
--> src/main.rs:4:18
  |
4 |     let f: u32 = File::open("hello.txt");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found error
  |
  = note: expected type `u32`
          found type `std::result::Result<std::fs::File, std::io::Error>`
```

This tells us the return type of the `File::open` function is a `Result<T, E>`. The type `T` has been filled in here with the type of the success value, `std::fs::File`. 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 to read from or write to. The function call also might fail: for example, the file name might not have permission to access the file. The `File::open` function needs to tell us whether it succeeded or failed and at the same time give us either the file handle or 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 a file handle. In the case where it fails, the value in `f` will be an instance of `std::io::Error`, which 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 what `File::open` returns. Listing 9-4 shows one way to handle the `Result` using a `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!("There was a problem opening the file: {:?}", error);
        }
    };
}
```

Listing 9-4: Using a `match` expression to handle the `Result` variants that `File::open` can return.

Note that, like the `Option` enum, the `Result` enum and its variants have been added to the standard prelude, so we don't need to specify `Result::` before the `Ok` and `Err` variants.

Here we tell Rust that when the result is `Ok`, return the inner `file` value or `Err`. We then assign that file handle value to the variable `f`. After the `match`, we can use `f` to do whatever we want with the file, like reading or writing.

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

```
thread 'main' panicked at 'There was a problem opening the file: Error { code: 2, message: "No such file or directory" }', src/main.rs:4:9
```

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. What if we want to take different actions for different failure reasons: if `File::open` failed because the file didn't exist, we want to create the file and return the handle to the new file. If `File::open` failed for some other reason—for example, because we didn't have permission to open the file—we want to handle that error differently. We can do this by adding more code to the `match`:

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(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => {
                    panic!(
                        "Tried to create file but there was a problem: {}",
                        e
                    )
                }
            }
        },
        Err(error) => {
            panic!(
                "There was a problem opening the file: {:?}",
                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`, 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 contains variants representing the different kinds of errors that might result from an operation. One variant we want to use is `ErrorKind::NotFound`, which indicates the file we're looking for doesn't exist yet.

The condition `if error.kind() == ErrorKind::NotFound` is called a *match guard*. It's a condition on a `match` arm that further refines the arm's pattern. This condition is needed so the code in the `Ok` arm is only run if the file exists. The `ref` in the pattern is needed so `error` is not moved into the guard and merely referenced by it. The reason you use `ref` to create a reference in a pattern will be covered in detail in Chapter 18. In short, in the context of a pattern, `& error` gives you its value, but `ref error` matches a value and gives you a reference to it.

The condition we want to check in the match guard is whether the value returned by `File::create` is the `NotFound` variant of the `ErrorKind` enum. If it is, we try to create the file again. However, because `File::create` could also fail, we need to add an inner `match` statement. When the file can't be opened, a different error message will be printed. The outer `match` stays the same so the program panics on any error besides the missing file.

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

Using `match` works well enough, but it can be a bit verbose and doesn't always catch all errors well. The `Result<T, E>` type has many helper methods defined on it to do common things like this. One of those methods, called `unwrap`, is a shortcut method that is implemented just like the `unwrap` statement we wrote in Listing 9-4. If the `Result` value is the `ok` variant, `unwrap` returns the value inside the `ok`. If the `Result` is the `Err` variant, `unwrap` will call the `panic!` function with the error message. 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 `unwrap` method:

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

Another method, `expect`, which is similar to `unwrap`, lets us also choose the message. Using `expect` instead of `unwrap` and providing good error messages intent and make tracking down the source of a panic easier. The syntax of `expect`:

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 an error message used by `expect` in its call to `panic!`. `expect` will be the parameter that starts with the text we specified, `Failed to open`, 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`, it's easier to find where in the code this error message is coming from. If we use `unwrap` in many places, it can take more time to figure out exactly which `unwrap` is causing the panic. It's better to use `expect` because multiple `unwrap` calls that panic print the same message.

Propagating Errors

When you're writing a function whose implementation calls something that might fail, if you're handling the error within this function, you can return the error to the calling function and let it decide what to do. This is known as *propagating* the error and gives more context to the error, so that you can see where there might be more information or logic that dictates how the error happened. You have more 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 it.

Filename: src/main.rs

```

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

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 manually in order to explore error handling; at the end, we'll show the easy return type of the function first: `Result<String, io::Error>`. This means the value of the type `Result<T, E>` where the generic parameter `T` has been filled by `String`, and the generic type `E` has been filled in with the concrete type `io::Error`. If the function succeeds without any problems, the code that calls this function will receive a `String` —the username that this function read from the file. If there are any problems, the code that calls this function will receive an `Err` value that contains more information about what the problems were. This is the return type of this function because that happens to be the type of the error value from both of the operations we're calling in this function's body that might fail: `File::open` and the `read_to_string` method.

The body of the function starts by calling the `File::open` function. Then we handle the value returned with a `match` similar to the `match` in Listing 9-4, only instead of the `Ok` case, we return early from this function and pass the error value from the calling code as this function's error value. If `File::open` succeeds, we store the variable `f` and continue.

Then we create a new `String` in variable `s` and call the `read_to_string` method on `f` to read the contents of the file into `s`. The `read_to_string` method also handles errors, so we don't need to do that ourselves. Because it might fail, even though `File::open` succeeded. So we need another `match`: 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 handles the failure of `File::open`. However, we don't need to explicitly say `return`, because the `match` expression is the return value of the function.

The code that calls this function will then handle getting either an `Ok` value that contains a `String` or an `Err` value that contains an `io::Error`. We don't know what the calling code will do with these values. If the calling code gets an `Err` value, it could call `panic!` and crash the program, or look up the username from somewhere other than a file, or do something else. We don't have enough information on what the calling code is actually trying to do with 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 `? 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 as it had in Listing 9-6, but this implementation uses the question mark operator.

Filename: src/main.rs

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

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 `?` placed after a `Result` value is defined to work in almost the same way as the expressions we defined to handle the `Result` values in Listing 9-6. If the value is `Ok`, the value inside the `Ok` will get returned from this expression, and the expression will return `Err` if the value is an `Err`, the `Err` will be returned from the whole function as if you 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 and Listing 9-7 does. In Listing 9-6, the `match` expression is taken by `?` go through the `from` function, defined in the `From` trait in the `std::error` module, which is used to convert errors from one type into another. When `?` calls the `from` function, the `Result` value 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 for many different reasons. As long as each error type implements the `From` trait, the `?` operator can automatically convert itself to the returned error type, `?` takes care of the conversion automatically.

In the context of Listing 9-7, the `?` at the end of the `File::open` call will return `Ok` to the variable `f`. If an error occurs, `?` will return early out of the whole function and return the `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 more concise. Listing 9-8 shows how Listing 9-7 could even shorten this code further by chaining method calls immediately after the `?`:

Filename: src/main.rs

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

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 `?`

We've moved the creation of the new `String` in `s` to the beginning of the function. Instead of creating a variable `f`, we've chained the call to `read_to_string` to the result of `File::open("hello.txt")?`. We still have a `?` at the end of the function, 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 identical to Listing 9-6 and Listing 9-7; this is just a different, more ergonomic way to write the function.

Speaking of different ways to write this function, there's a way to make this even easier.

Filename: `src/main.rs`

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

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

Listing 9-9: Using `fs::read_to_string`

Reading a file into a string is a fairly common operation, and so Rust provides a standard library function called `fs::read_to_string` that will open the file, create a new `String` from the contents of the file, and put the contents into that `String`, and then return it. Of course, this function also has the opportunity to show off all of this error handling, so we did it the hard way.

The `?` Operator Can Only Be Used in Functions That Return `Result`

The `?` operator can only be used in functions that have a return type of `Result<_, _>` defined to work in the same way as the `match` expression we defined in Listing 9-7. This means that the `match` expression requires a return type of `Result<_, Err(e)>`, so the return type of the function must be a `Result` to be compatible with this `return`.

Let's look at what happens if we use `?` in the `main` function, which you'll recall from Listing 9-1 looks like this:

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

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

When we compile this code, we get the following error message:

```
error[E0277]: the trait bound `(): std::ops::Try` is not satisfied
--> src/main.rs:4:13
  |
4 |     let f = File::open("hello.txt")?;
  |     -----
  |     |
  |     the `?` operator can only be used in a function that returns
  |     `Result` (or another type that implements `std::ops::Try`)
  |     in this macro invocation
  |
= help: the trait `std::ops::Try` is not implemented for `()`
= note: required by `std::ops::Try::from_error`
```

This error points out that we're only allowed to use `?` in a function that returns `Result` if the function itself doesn't return `Result`, when you call other functions that return `Result`.

match or one of the Result methods to handle the Result instead of using propagate the error to the calling code.

Now that we've discussed the details of calling panic! or returning Result, 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 code panics, there's no way to recover. You could call panic! for any errors where there's a possible way to recover or not, but then you're making the decision to call panic! in your code that a situation is unrecoverable. When you choose to return Result, you're giving the calling code options rather than making the decision for it. The calling code can attempt to recover in a way that's appropriate for its situation, or it could decide that the attempt to recover 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 unless your function that might fail.

In rare situations, it's more appropriate to write code that panics instead of returning an error. We'll explore why it's appropriate to panic in examples, prototype code, and tests. There are also situations in which the compiler can't tell that failure is impossible, but you as a developer will know better. This chapter will conclude with some general guidelines on how to decide whether to panic or return an error from your code.

Examples, Prototype Code, and Tests

When you're writing an example to illustrate some concept, having robust error handling code as well can make the example less clear. In examples, it's understood that any method like unwrap that could panic is meant as a placeholder for the way your application will handle errors, which can differ based on what the rest of your application does.

Similarly, the unwrap and expect methods are very handy when prototyping code to decide how to handle errors. They leave clear markers in your code for when you want 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 is part of the functionality under test. Because panic! is how a test is marked as a failure, 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 guarantees that the Result will have an ok value, but the logic isn't something the compiler understands. For example, if you have a Result value that you need to handle: whatever operation you're calling on it has a non-zero possibility of failing in general, even though it's logically impossible in your particular case. You can ensure by manually inspecting the code that you'll never have an Err value, so it's 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 valid IP address, so it's acceptable to use `unwrap` here. However, having a `hardcoded` doesn't change the return type of the `parse` method: we still get a `Result`. This will still make us handle the `Result` as if the `Err` variant is a possibility because the parser is smart enough to see that this string is always a valid IP address. If the IP address was user rather than being hardcoded into the program and therefore *did* have an error, 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 reach a state that's not supposed to happen. In this context, a *bad state* is when some assumption, guarantee, contract, or invariant is 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 not something that's *expected* to happen occasionally.
- Your code after this point needs to rely on not being in this bad state.
- There's not a good way to encode this information in the types you use.

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

When a bad state is reached, but it's expected to happen no matter how well the code is written, it's still more appropriate to return a `Result` rather than to make a `panic!` call. For example, if you're writing a JSON parser being given malformed data or an HTTP request returning a status that's outside the allowed range or exceeding a rate limit. In these cases, you should indicate that failure is an expected part of the normal behavior of the system by returning a `Result` to propagate these bad states upward so the calling code can decide what to do with them. To call `panic!` wouldn't be the best way to handle these cases.

When your code performs operations on values, your code should verify the inputs are valid before proceeding. This is mostly for safety reasons: attempting to access memory that doesn't belong to your code can expose your code to vulnerabilities. This is the main reason the standard library uses `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. Functionality like this is only guaranteed if the inputs meet particular requirements. For example, if a function's contract is violated, it makes sense because a contract violation always indicates that the function's behavior is undefined. It's not a kind of error you want the calling code to have to explicitly handle. A reasonable way for calling code to recover from an error is to provide a reasonable way for calling code to recover; the calling *programmers* need to understand what the function does and how it handles errors. For example, especially when a violation will cause a panic, should be explained in the function's documentation.

However, having lots of error checks in all of your functions would be verbose and repetitive. Fortunately, you can use Rust's type system (and thus the type checking the compiler) to handle many of the checks for you. If your function has a particular type as a parameter, the compiler knows that the function expects to receive that type. For example, if you have a type rather than an `Option`, your program expects to receive a value of that type rather than *nothing*. Your code then doesn't have to handle two cases for the `Some` variant: the `None` variant will only have one case for definitely having a value. Code trying to pass `None` to a function that expects `Some` won't even compile, so your function doesn't have to check for that case at runtime. For example, if you're writing a function that takes an unsigned integer type such as `u32`, which ensures the value is non-negative.

Let's take the idea of using Rust's type system to ensure we have a valid value and look at creating a custom type for validation. Recall the guessing game in Chapter 10.

code asked the user to guess a number between 1 and 100. We never validate if the guess was between those numbers before checking it against our secret number; if the guess was positive. In this case, the consequences were not very dire: only “Too low” would still be correct. But it would be a useful enhancement to give users different behavior when a user guesses a number that's outside the user types, for example, letters instead.

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

```
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 a message, and calls `continue` to start the next iteration of the loop and ask for another guess. Once we're inside the loop, if the `match` expression, we can proceed with the comparisons between `guess` and the `secret_number`. If `guess` is between 1 and 100, we can proceed with the comparisons between `guess` and the `secret_number`.

However, this is not an ideal solution: if it was absolutely critical that the program checked for values between 1 and 100, and it had many functions with this requirement, then this code would need to be repeated in every function (and might impact performance).

Instead, we can make a new type and put the validations in a function to create a new type rather than repeating the validations everywhere. That way, it's safe for us to use the new type in their signatures and confidently use the values they receive. Let's list how to define a `Guess` type that will only create an instance of `Guess` if the value is between 1 and 100:

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> u32 {
        self.value
    }
}
```

Listing 9-10: A `Guess` type that will only continue with values between 1 and

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

Then we implement an associated function named `new` on `Guess` that creates values. The `new` function is defined to have one parameter named `value` of type `u32`. 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 user to the fact that they're writing the calling code that they have a bug they need to fix, because creating a `value` outside this range would violate the contract that `Guess::new` is relying on. Which `Guess::new` might panic should be discussed in its public-facing API documentation. Cover documentation conventions indicating the possibility of a `panic!` in the code you create in Chapter 14. If `value` does pass the test, we create a new `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 parameters, and returns a `u32`. This kind of method is sometimes called a `getter`. Its purpose is to get some data from its fields and return it. This public method returns the `value` field of the `Guess` struct is private. It's important that the `value` field of the `Guess` struct is not allowed to set `value` directly: code outside the `value` field of the `Guess` struct to create an instance of `Guess`, thereby ensuring that the `value` field has been checked by the conditions in the `Guess::new` function.

A function that has a parameter or returns only numbers between 1 and 100 has its signature that it takes or returns a `Guess` rather than a `u32` and wouldn't need additional checks in its body.

Summary

Rust's error handling features are designed to help you write more robust code. They signal that your program is in a state it can't handle and lets you tell the program what to do instead of trying to proceed with invalid or incorrect values. The `Result` enum uses `Ok` and `Err` variants to indicate that operations might fail in a way that your code could recover from. Telling code that calls your code that it needs to handle potential success or failure gracefully with `panic!` and `Result` in the appropriate situations will make your code more robust and less prone to inevitable problems.

Now that you've seen useful ways that the standard library uses generics without you having to write them yourself, we'll talk about how generics work and how you can use them in your own code.

Generic Types, Traits, and Lifetimes

Every programming language has tools for effectively handling the duplication of code. One such tool is *generics*. Generics are abstract stand-ins for concrete types (types that have specific values). When we're writing code, we can express the behavior of generics or how they should be used without knowing what will be in their place when compiling and running the code.

Similar to the way a function takes parameters with unknown values to run them multiple times, functions can take parameters of some generic type. For example, a type like `i32` or `String`. In fact, we've already used generics in Chapter 6 when we talked about `Vec<T>` and `HashMap<K, V>`, and Chapter 9 with `Result<T, E>`. In this chapter, we'll learn how to define your own types, functions, and methods with generics!

First, we'll review how to extract a function to reduce code duplication. Next, we'll learn how to make a generic function from two functions that differ only in 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 use generic types to constrain a generic type to only those types that have a particular trait, 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 borrow values in many ways, enabling the compiler to check that the references are valid.

Removing Duplication by Extracting a Function

Before diving into generics syntax, let's first look at how to remove duplication by extracting a function. Then we'll apply this technique to extracting the same code in the same way that you recognize duplicated code to extract into a function. You'll learn how to recognize duplicated code that can use generics.

Consider a short program that finds the largest number in a list, as shown in Listing 10-1.

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: Code to find the largest number in a list of numbers

This code stores a list of integers in the variable `number_list` and places the largest value in a variable named `largest`. Then it iterates through all the numbers in the list. If the current number is greater than the number stored in `largest`, it replaces the number in `largest`. However, if the current number is less than the largest number seen so far, it does not 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.

To find the largest number in two different lists of numbers, we can duplicate 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 will code in multiple places when we want to change it.

To eliminate this duplication, we can create an abstraction by defining a function that takes any list of integers given to it in a parameter. This solution makes our code easier to express the concept of finding the largest number in a list abstractly.

In Listing 10-3, we extracted the code that finds the largest number into a function named `largest`. Unlike the code in Listing 10-1, which can find the largest number in only one list, this program can find the largest number in two different lists.

Filename: src/main.rs

```

fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        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 `Vec` values that we might pass into the function. As a result, when we call the function, we pass in the specific values that we want.

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 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 a similar way that the function body can operate on an abstract `list` instead of concrete types. Generics allow code to operate on abstract types.

For example, say we had two functions: one that finds the largest item in a `Vec` and one that finds the largest item in a slice of `char` values. How would we eliminate code duplication? Let's find out!

Generic Data Types

We can use generics to create definitions for items like function signatures and then reuse them with many different concrete data types. Let's first look at how to define generic functions, enums, and methods using generics. Then we'll discuss how generics affect code organization.

In Function Definitions

When defining a function that uses generics, we place the generics in the signature where we would usually specify the data types of the parameters and return value. This makes our code more flexible and provides more functionality to callers of our function without introducing code duplication.

Continuing with our `largest` function, Listing 10-4 shows two functions that both find the largest value in a slice.

Filename: `src/main.rs`

```

fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        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 th

The `largest_i32` function is the one we extracted in Listing 10-3 that finds t slice. The `largest_char` function finds the largest `char` in a slice. The funct same code, so let's eliminate the duplication by introducing a generic type p function.

To parameterize the types in the new function we'll define, we need to name just as we do for the value parameters to a function. You can use any identif name. But we'll use `T` because, by convention, parameter names in Rust are letter, and Rust's type-naming convention is CamelCase. Short for "type," `T` i most Rust programmers.

When we use a parameter in the body of the function, we have to declare th the signature so the compiler knows what that name means. Similarly, when parameter name in a function signature, we have to declare the type param use it. To define the generic `largest` function, place type name declaration: `<>`, 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 parameter named `list`, which is a slice of values of type `T`. The `largest` f value of the same type `T`.

Listing 10-5 shows the combined `largest` function definition using the gen signature. The listing also shows how we can call the function with either a s

char values. Note that this code won't compile yet, but we'll fix it later in thi

Filename: src/main.rs

```
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        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: A definition of the `largest` function that uses generic type parameters. This code won't compile yet.

If we compile this code right now, we'll get this error:

```
error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
  |
5 |         if item > largest {
  |         ^^^^^^^^^^^^^^
  |
= note: an implementation of `std::cmp::PartialOrd` might be missing
```

The note mentions `std::cmp::PartialOrd`, which is a *trait*. We'll talk about traits in Chapter 17. For now, this error states that the body of `largest` won't work for all possible types. Because we want to compare values of type `T` in the body, we can only implement `PartialOrd` if `T` 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 can implement this trait on any type that has a particular trait in the "Trait Bounds" section, but let's start by looking at some 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. Listing 10-6 shows how to define a `Point<T>` struct to hold `x` and `y` coordinates 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 functions: we declare the name of the type parameter inside angle brackets just after the struct keyword. Then we can use the generic type in the struct definition where we would normally use concrete data types.

Note that because we've used only one generic type to define `Point<T>`, this `Point<T>` struct is generic over some type `T`, and the fields `x` and `y` are both of type `T`. If we create an instance of a `Point<T>` that has different types for `x` and `y` 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 type `T`.

In this example, when we assign the integer value 5 to `x`, we let the compiler know that `x` will be an integer for this instance of `Point<T>`. Then when we specify that `y` has the type 4.0, we've defined to have the same type as `x`, we'll get a type mismatch error like this:

```

error[E0308]: mismatched types
--> src/main.rs:7:38
 |
7 |     let wont_work = Point { x: 5, y: 4.0 };
|                                ^^^ expected integral variable
|                                floating-point variable
|
= note: expected type `'{integer}`
        found type `'{float}'
```

To define a `Point` struct where `x` and `y` are both generics but could have different types, we could use multiple generic type parameters. For example, in Listing 10-8, we can change the `Point` struct 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

Now all the instances of `Point` shown are allowed! You can use as many generic types in your code as you want, but using more than a few makes your code hard to read. If you have lots of generic types in your code, it could indicate that your code needs refactoring or some other pieces.

In Enum Definitions

As we did with structs, we can define enums to hold generic data types in them. Let's take another look at the `Option<T>` enum that the standard library provides, which is defined in Listing 6:

```

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

```

This definition should now make more sense to you. As you can see, `Option` is generic over type `T` and has two variants: `Some`, which holds one value of type `T`, and `None`, which is a variant that doesn't hold any value. By using the `Option<T>` enum, we can express the concept of having an optional value, and because `Option<T>` is generic, we can use it with any type of value, no matter what the type of the optional value is.

Enums can use multiple generic types as well. The definition of the `Result` enum from 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 easy to use `Result` anywhere we have an operation that might succeed (returning a value of type `T`) or fail (return an error of some type `E`). In fact, this is what we used to do when reading files: 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 enums like `Option` and `Result`.

In Method Definitions

We can implement methods on structs and enums (as we did in Chapter 5) *à la* their definitions, too. Listing 10-9 shows the `Point<T>` struct we defined in Listing 10-8 and the 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 returns 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 `x` field `x`.

Note that we have to declare `T` just after `impl` so we can use it to specify the type of the return value of the methods on the type `Point<T>`. By declaring `T` as a generic type after `impl`, the type in the angle brackets in `Point` is a generic type rather than a concrete type.

We could, for example, implement methods only on `Point<f32>` instances rather than instances with any generic type. In Listing 10-10 we use the concrete type `f32` to declare any types after `impl`.

```
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 generic type parameter `T`

This code means the type `Point<f32>` will have a method named `distance_from_origin`. Instances of `Point<T>` where `T` is not of type `f32` will not have this method. This method measures how far our point is from the point at coordinates (0.0, 0.0) and uses floating-point operations that are available only for floating-point types.

Generic type parameters in a struct definition aren't always the same as those in the struct's method signatures. For example, Listing 10-11 defines the method `mixup` on the `Point<T, U>` struct from Listing 10-8. The method takes another `Point` as an argument, but the type of that `Point` might have different types than the `self` `Point` we're calling `mixup` on. The `mixup` method takes a `Point` instance with the `x` value from the `self` `Point` (of type `T`) and the `y` value from the passed-in `Point` (of type `U`).

Filename: `src/main.rs`

```

struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        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 different generic types than its struct's def

In `main`, we've defined a `Point` that has an `i32` for `x` (with value `5`) and a `10.4`. The `p2` variable is a `Point` struct that has a string slice for `x` (with `v` `char` for `y` (with value `c`). Calling `mixup` on `p1` with the argument `p2` gives an `i32` for `x`, because `x` came from `p1`. The `p3` variable will have a `char` 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 `T` and `U` are declared with `impl` and some are declared with the method definition. Here `T` and `U` are declared after `impl`, because they go with the struct. The generic parameters `V` and `W` are declared after `fn mixup`, because they're part of the method.

Performance of Code Using Generics

You might be wondering whether there is a runtime cost when you're using generic parameters. The good news is that Rust implements generics in such a way that it won't run any slower using generic types than it would with concrete types.

Rust accomplishes this by performing monomorphization of the code that is done at compile time. *Monomorphization* is the process of turning generic code into specialized code for the concrete types that are used when compiled.

In this process, the compiler does the opposite of the steps we used to create generic code. Listing 10-5: the compiler looks at all the places where generic code is called and creates specialized code for each of the concrete types the generic code is called with.

Let's look at how this works with an example that uses the standard library's `Option` type.

```

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 that one is `i32` and the other is `f64`. As such, it expands the generic definition into two separate pieces of code:

Option_i32 and Option_f64, thereby replacing the generic definition with:

The monomorphized version of the code looks like the following. The generic definitions are replaced with the specific definitions created by the compiler:

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);
}
```

Because Rust compiles generic code into code that specifies the type in each generic definition, there is 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 generic code extremely efficient at runtime.

Traits: Defining Shared Behavior

A *trait* tells the Rust compiler about functionality a particular type has and can implement. We can use traits to define shared behavior in an abstract way. We can then specify that a generic can be any type that has certain behavior.

Note: Traits are similar to a feature often called *interfaces* in other languages, but there are some differences.

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types can have different behaviors. A trait defines the behavior of a type by specifying method signatures together to define a set of behaviors necessary to accomplish something.

For example, let's say we have multiple structs that hold various kinds of data. We might have a NewsArticle struct that holds a news story filed in a particular location and a Tweet struct that holds a tweet with a timestamp, a user ID, and a message. Both types have at most 280 characters along with metadata that indicates whether it was a reply to another tweet or a reply to another tweet.

We want to make a media aggregator library that can display summaries of the data stored in a NewsArticle or Tweet instance. To do this, we need a Summary trait that defines the behavior of a type that can be summarized. We can then implement the definition of a 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, `Summary`. Inside the curly brackets, we declare the method signatures that describe the behavior that types that implement this trait must provide. In this case is `fn summarize(&self) -> String`.

After the method signature, instead of providing an implementation within curly braces, we end the definition with a semicolon. Each type implementing this trait must provide its own custom behavior for the `summarize` method. The compiler will enforce that any type that has the `Summary` trait implemented must have a `summarize` defined with this signature exactly.

A trait can have multiple methods in its body: the method signatures are listed one after another, separated by line ends in a semicolon.

Implementing a Trait on a Type

Now that we've defined the desired behavior using the `Summary` trait, we can implement it on specific types in our media aggregator. Listing 10-13 shows an implementation of the `Summary` trait for the `NewsArticle` and `Tweet` types. The `NewsArticle` struct that uses the headline, the author, and the location to construct a summary. For the `Tweet` struct, we define `summarize` as the username followed by the tweet content, 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. After `impl`, we put the trait name that we want to implement, then use the `for` keyword to specify the name of the type we want to implement the trait for. Within the `impl` block, we implement the trait methods for the type.

method signatures that the trait definition has defined. Instead of adding a signature, we use curly brackets and fill in the method body with the specific methods of the trait to have for the particular type.

After implementing the trait, we can call the methods on instances of `NewsArticle` the same way we call regular methods, like this:

```
let tweet = Tweet {  
    username: String::from("horse_ebooks"),  
    content: String::from("of course, as you probably already know"),  
    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,
```

Note that because we defined the `Summary` trait and the `NewsArticle` and `lib.rs` in Listing 10-13, they're all in the same scope. Let's say this `lib.rs` is for a aggregator and someone else wants to use our crate's functionality to implement on a struct defined within their library's scope. They would need to import the first. They would do so by specifying `use aggregator::Summary;`, which the implement `Summary` for their type. The `Summary` trait would also need to be another crate to implement it, which it is because we put the `pub` keyword in Listing 10-12.

One restriction to note with trait implementations is that we can implement either the trait or the type is local to our crate. For example, we can implement traits like `Display` on a custom type like `Tweet` as part of our aggregator because the type `Tweet` is local to our aggregator crate. We can also implement `Vec<T>` in our aggregator crate, because the trait `Summary` is local to our 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` already exists in the standard library and aren't local to our aggregator crate. This restricts programs called *coherence*, and more specifically the *orphan rule*, so named because the type is not present. This rule ensures that other people's code can't break your code. Without the rule, two crates could implement the same trait for the same type, and you 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 for a particular type, we can keep or override each method's default behavior.

Listing 10-14 shows how to specify a default string for the `summarize` method 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: Definition of a `Summary` trait with a default implementation of

To use a default implementation to summarize instances of `NewsArticle` in custom implementation, we specify an empty `impl` block with `impl Summary`.

Even though we're no longer defining the `summarize` method on `NewsArticle`, we provided a default implementation and specified that `NewsArticle` implements it. As a result, we can still call the `summarize` method on an instance of `NewsArticle`:

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Champions"),
    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 for `summarize` doesn't require us to change the implementation of `Summary` on `Tweet` in Listing 10-13. The reason is that the default implementation is the same as the syntax for implementing a trait method with a default implementation.

Default implementations can call other methods in the same trait, even if they have a default implementation. In this way, a trait can provide a lot of useful functionality without requiring implementors to specify a small part of it. For example, we could define a `summarize_author` method whose implementation is required, and then have 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` via an `impl` block for 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 `Tweet`. The default implementation of `summarize` will call the definition of `summarize_author` provided. Because we've implemented `summarize_author`, the `Summary` trait provides 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"),
    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 method on the same method.

Traits as arguments

Now that you know how to define traits and implement those traits on types, we can use traits to accept arguments of many different types.

For example, in Listing 10-13, we implemented the `Summary` trait on the type `NewsArticle`. We can define a function `notify` that calls the `summarize` method on any type which is of some type that implements the `Summary` trait. To do this, we can use trait syntax, like this:

```

pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

```

In the body of `notify`, we can call any methods on `item` that come from the `Summary` trait, like `summarize`.

Trait Bounds

The `impl Trait` syntax works for short examples, but is syntax sugar for a longer form called a 'trait bound', and it looks like this:

```

pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}

```

This is equivalent to the example above, but is a bit more verbose. We place the declaration of the generic type parameter, after a colon and inside angle brackets. Since we have a trait bound on `T`, we can call `notify` and pass in any instance of `Summary`. This means we can call `notify` on any type that implements `Summary`. If we try to call `notify` on a type that doesn't implement `Summary`, the compiler will complain.

When should you use this form over `impl Trait`? While `impl Trait` is nice for simple traits, trait bounds are nice for more complex ones. For example, say we wanted to implement `Summary` for multiple types:

```

pub fn notify(item1: impl Summary, item2: impl Summary) {
    println!("Multiple items! {} {}", item1.summarize(), item2.summarize());
}

```

The version with the bound is a bit easier. In general, you should use whatever form makes the most understandable code.

Multiple trait bounds with +

We can specify multiple trait bounds on a generic type using the `+` syntax. For example, we could add `Display` and `Clone` to the `Summary` trait. This would mean that `Summary` must implement both `Display` and `Clone`. As you might expect, this grows quite complex!

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U)
```

where clauses for clearer code

However, there are downsides to using too many trait bounds. Each generic type needs to implement all of its trait bounds, so functions with multiple generic type parameters can have lots of trait bounds between a function's name and its parameter list, making the function signature very long. For this reason, Rust has alternate syntax for specifying trait bounds inside a `where` clause. So instead of writing this:

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 in that the function name, parameters, and trait bounds are close together, similar to a function without lots of trait bounds.

Returning Traits

We can use the `impl Trait` syntax in return position as well, to return something that implements a trait:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know"),
        reply: false,
        retweet: false,
    }
}
```

This signature says, "I'm going to return something that implements the `Summary` trait." It's not going to tell you the exact type. In our case, we're returning a `Tweet`, but the compiler knows that.

Why is this useful? In chapter 13, we're going to learn about two features that make this useful: closures, and iterators. These features create types that only the compiler knows about, and they can be very, very long. `impl Trait` lets you simply say "this returns an `Iterator`" without having to write out a really long type.

This only works if you have a single type that you're returning, however. For example:

```

fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from("Penguins win the Stanley Cup CI
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from("The Pittsburgh Penguins once ag
            hockey team in the NHL."),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably alre
            reply: false,
            retweet: false,
        }
    }
}

```

Here, we try to return either a `NewsArticle` or a `Tweet`. This cannot work, c around how `impl Trait` works. To write this code, you'll have to wait until C objects".

Fixing the `largest` Function with Trait Bounds

Now that you know how to specify the behavior you want to use using the generic bounds, let's return to Listing 10-5 to fix the definition of the `largest` function with type parameter! Last time we tried to run that code, we received this error:

```

error[E0369]: binary operation `>` cannot be applied to type `T`
--> src/main.rs:5:12
  |
5 |         if item > largest {
  |         ^^^^^^^^^^^^^^
  |
= note: an implementation of `std::cmp::PartialOrd` might be mis:

```

In the body of `largest` we wanted to compare two values of type `T` using the `>` operator. Because that operator is defined as a default method on the standard trait `std::cmp::PartialOrd`, we need to specify `PartialOrd` in the trait bounds so that the `largest` function can work on slices of any type that we can compare. We don't need to bring it 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:

```

error[E0508]: cannot move out of type `[T]`, a non-copy slice
--> src/main.rs:2:23
|           let mut largest = list[0];
|           ^^^^^^
|           |
|           cannot move out of here
|           help: consider using a reference instead

error[E0507]: cannot move out of borrowed content
--> src/main.rs:4:9
|           for &item in list.iter() {
|           ^----_
|           ||
|           |hint: to prevent move, use `ref item` or `ref mut item`
|           cannot move out of borrowed content

```

The key line in this error is `cannot move out of type [T]`, a non-copy slice. In generic versions of the `largest` function, we were only trying to find the largest value in a slice. As discussed in the “Stack-Only Data: Copy” section in Chapter 4, types like `i32` have a known size and can be stored on the stack, so they implement the `Copy` trait. But because the `largest` function is generic, it became possible for the `list` parameter to have ownership of the values in the slice. Consequently, we wouldn’t be able to move the values into the `largest` variable, resulting in this error.

To call this code with only those types that implement the `Copy` trait, we can add a bound to `T`. Listing 10-15 shows the complete code of a generic `largest` function that works 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.iter() {
        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 an array slice as long as the elements implement the `PartialOrd` and `Copy` traits

If we don’t want to restrict the `largest` function to the types that implement `Copy`, we could specify that `T` has the trait bound `Clone` instead of `Copy`. Then we can move the values from the slice when we want the `largest` function to have ownership. Using the `clone` method on the slice will make a copy of the slice, leaving the original slice owned by the caller.

we're potentially making more heap allocations in the case of types that own large amounts of data, and heap allocations can be slow if we're working with large amounts of data.

Another way we could implement `largest` is for the function to return a reference to the slice. If we change the return type to `&T` instead of `T`, thereby changing the return type to a reference, we wouldn't need the `Clone` or `Copy` trait bounds and won't need to make heap allocations. Try implementing these alternate solutions on your own!

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, in Listing 10-16 always implements the `new` function. But `Pair<T>` only implements the `cmp_display` method if its inner type `T` implements the `PartialOrd` trait, which means it must implement the `Display` trait that enables printing.

```
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. For example, 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 `ToString` trait on any type that implements the `Display` trait has a `ToString` 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 `ToString` method defined by the `ToString` trait on any type that implements the `Display` trait. This will turn integers into their corresponding `String` values like this because integers implement the `Display` trait:

```
let s = 3.to_string();
```

Blanket implementations appear in the documentation for the trait in "Implementing Traits".

Traits and trait bounds let us write code that uses generic type parameters to specify what behavior we want. We can then use the trait bound information to check that all the concrete types provide the correct behavior. In dynamically typed languages, we would get called a method on a type that the type didn't implement. But Rust moves the time so we're forced to fix the problems before our code is even able to run. We have to write code that checks for behavior at runtime because we've already done so. Doing so improves performance without having to give up the flexibility.

Another kind of generic that we've already been using is called *lifetimes*. Rather than specifying the behavior we want, lifetimes ensure that references are valid as long as the reference is. Let's look at how lifetimes do that.

Validating References with Lifetimes

One detail we didn't discuss in the "References and Borrowing" section in Chapter 8 is that every reference in Rust has a *lifetime*, which is the scope for which that reference is valid. Lifetimes are implicit and inferred, just like most of the time, types are inferred. Rust requires explicit lifetime annotations when multiple types are possible. In a similar way, we must annotate lifetime parameters to ensure the actual relationships of references could be related in a few different ways. Rust requires explicit lifetime parameters to ensure the actual references are definitely valid.

The concept of lifetimes is somewhat different from tools in other programming languages, making lifetimes Rust's most distinctive feature. Although we won't cover lifetimes in this chapter, we'll discuss common ways you might encounter lifetime syntax and be familiar with the concepts. See the "Advanced Lifetimes" section in Chapter 1 for more information.

Preventing Dangling References with Lifetimes

The main aim of lifetimes is to prevent dangling references, which cause a program to access data other than the data it's intended to reference. Consider the program in Listing 10-17:

```
{
    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 with initial values, so the variable name exists in the outer scope. At first glance,

be in conflict with Rust's having no null values. However, if we try to use a it a value, we'll get a compile-time error, which shows that Rust indeed do 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 print the value of `r`. Then the inner scope ends, and we attempt to print the value of `r`. The code won't compile because the value of `r` is referring to has gone out of scope before the inner scope ends. Here is the error message:

```
error[E0597]: `x` does not live long enough
--> src/main.rs:7:5
   |
6 |         r = &x;
   |         - borrow occurs here
7 |     }
   |     ^ `x` dropped here while still borrowed
...
10| }
   | - borrowed value needs to live until here
```

The variable `x` doesn't "live long enough." The reason is that `x` will be out of scope ends on line 7. But `r` is still valid for the outer scope; because its scope "lives longer." If Rust allowed this code to work, `r` would be referencing memory that has already been deallocated when `x` went out of scope, and anything we tried to do with `r` would be undefined behavior. So how does Rust determine that this code is invalid? It uses a borrow check.

The Borrow Checker

The Rust compiler has a *borrow checker* that compares scopes to determine if the code is 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`'

Here, we've annotated the lifetime of `r` with '`'a`' and the lifetime of `x` with '`'b`'. The inner '`'b`' block is much smaller than the outer '`'a`' lifetime block. At compile time, the borrow checker compares the size of the two lifetimes and sees that `r` has a lifetime of '`'a`' but that its lifetime is '`'b`'. The program is rejected because '`'b`' is shorter than '`'a`: the reference to `r` doesn't live as long as the reference.

Listing 10-19 fixes the code so it doesn't have a dangling reference and compiles successfully.

```
{
    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 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 references, we can see why references will always be valid, let's explore generic lifetimes of parameters in the context of functions.

Generic Lifetimes in Functions

Let's write a function that returns the longer of two string slices. This function takes two string slices and return a string slice. After we've implemented the `longest` function, 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 string.

Note that we want the function to take string slices, which are references, because the `longest` function does not own its parameters. We want to allow the function to take slices of a `String` (the type stored in the variable `string1`) as well as string literals (the type stored in the variable `string2`).

Refer to the “String Slices as Parameters” section in Chapter 4 for more discussion. 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 work.

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 string, but does not yet compile.

Instead, we get the following error that talks about lifetimes:

```
error[E0106]: missing lifetime specifier
--> src/main.rs:1:33
|
1 | fn longest(x: &str, y: &str) -> &str {
|                                         ^ expected lifetime parameter
|
|= help: this function's return type contains a borrowed value, but its
signature does not say whether it is borrowed from `x` or `y`
```

The help text reveals that the return type needs a generic lifetime parameter to tell whether the reference being returned refers to `x` or `y`. Actually, we don't know which reference the return value will be borrowed from 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 we pass to `x` and `y`, 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 code in Listings 10-18 and 10-19 to determine whether the reference we return will be borrowed from `x` or `y`. The borrow checker can't determine this either, because it doesn't know how `x` and `y` relate to the lifetime of the return value. To fix this error, we'll add generic lifetime annotations to the parameters and the return type that define the relationship between the references so the borrow checker can figure out what's going on.

Lifetime Annotation Syntax

Lifetime annotations don't change how long any of the references live. Just as with regular type annotations, you can add them to any type when the signature specifies a generic type parameter, functions can have multiple lifetime parameters associated with any lifetime by specifying a generic lifetime parameter. Lifetime annotations let you express relationships of the lifetimes of multiple references to each other without affecting the actual values of the references.

Lifetime annotations have a slightly unusual syntax: the names of lifetime parameters begin with an apostrophe ('') and are usually all lowercase and very short, like `'a`. You can also use the name `'a`. We place lifetime parameter annotations after the `&` or `&mut` keyword 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 a reference to an `i32` that has a lifetime parameter named `'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 compiler needs more information to know how generic lifetime parameters of multiple references relate to each other. Let's say we have a function with the parameter `first` that is a reference to a string. The function also has another parameter named `second` that is another reference to a string. Both references also have the lifetime `'a`. The lifetime annotations indicate that the references 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. When we declared the parameters, we need to declare generic lifetime parameters inside angle brackets. We do this by placing the lifetime parameter name inside the angle brackets after the function name and the parameter list. The constraint we want to express is that the lifetimes of the references in the parameters and the return value must have the same lifetime `'a`. We can 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 must have the same lifetime `'a`

This code should compile and produce the result we want when we use it with Listing 10-20.

The function signature now tells Rust that for some lifetime `'a`, the function both of which are string slices that live at least as long as lifetime `'a`. The function tells Rust that the string slice returned from the function will live at least as long as `'a`. These constraints are what we want Rust to enforce. Remember, when we specify parameters in this function signature, we're not changing the lifetimes of anything returned. Rather, we're specifying that the borrow checker should reject any attempt to adhere to these constraints. Note that the `longest` function doesn't need to know that `x` and `y` will live, only that some scope can be substituted for `'a` that will satisfy the constraint.

When annotating lifetimes in functions, the annotations go in the function signature, not in the function body. Rust can analyze the code within the function without any help. If the function has references to or from code outside that function, it becomes difficult to figure out the lifetimes of the parameters or return values on its own. The lifetimes are different each time the function is called. This is why we need to annotate them.

When we pass concrete references to `longest`, the concrete lifetime that is used is the part of the scope of `x` that overlaps with the scope of `y`. In other words, `longest` will get the concrete lifetime that is equal to the smaller of the lifetimes of `x` and `y`. If we annotated the returned reference with the same lifetime parameter `'a`, the code would 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 references with 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 with 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. If you run this code, and you'll see that the borrow checker approves of this code; it will print out `The longest string is long`.

Next, let's try an example that shows that the lifetime of the reference in `result` is smaller than the lifetime of the two arguments. We'll move the declaration of the `result` variable into the inner scope but leave the assignment of the value to the `result` variable inside the `string2`. Then we'll move the `println!` that uses `result` outside 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'll get this error:

```
error[E0597]: `string2` does not live long enough
--> src/main.rs:15:5
 |
14 |         result = longest(string1.as_str(), string2.as_str());           ----- borrow occurs here
15 |         }
16 |         ^ `string2` dropped here while still borrowed
17 |     println!("The longest string is {}", result);
| - borrowed value needs to live until here
```

The error shows that for `result` to be valid for the `println!` statement, `string2` must be valid until the end of the outer scope. Rust knows this because we annotated the function parameters and return values using the same lifetime parameter, `'`.

As humans, we can look at this code and see that `string1` is longer than `string2`, so `result` will contain a reference to `string1`. Because `string1` has not gone out of scope, the reference to `string1` will still be valid for the `println!` statement. However, the `longest` function returns a reference to the shorter string, so the reference to `string2` is valid in this case. We've told Rust that the lifetime of the reference returned by `longest` is the same as the lifetime of the argument `string2`. Therefore, the borrow checker disallows the code in Listing 10-24 as possibly causing a data race.

Try designing more experiments that vary the values and lifetimes of the references returned by the `longest` function and how the returned reference is used. Make hypotheses about what happens in each case, and then write code to test your hypotheses. If your experiments will pass the borrow checker before you compile; then check the results against your hypotheses.

Thinking in Terms of Lifetimes

The way in which you need to specify lifetime parameters depends on what you're trying to do. For example, if we changed the implementation of the `longest` function to return a copy of the longer string slice rather than the longest string slice, we wouldn't need to specify a lifetime parameter. The following code will compile:

Filename: src/main.rs

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

In this example, we've specified a lifetime parameter `'a` for the parameter `x` but not for the parameter `y`, because the lifetime of `y` does not have any relationship to the lifetime of `x` or the return value.

When returning a reference from a function, the lifetime parameter for the return value must match the lifetime parameter for one of the parameters. If the reference returned by the function does not match the lifetime of one of the parameters, it must refer to a value created within this function, which would result in a dangling reference because the value will go out of scope at the end of the function. Here is an 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, the code will fail to compile because the return value lifetime is not related to the lifetime of the parameter `x`. Here is the error message we get:

```
error[E0597]: `result` does not live long enough
--> src/main.rs:3:5
  |
3 |     result.as_str()
  |     ^^^^^^ does not live long enough
4 | }
  |   - borrowed value only lives until here
  |
note: borrowed value must be valid for the lifetime 'a as defined on the
function body at 1:1...
--> src/main.rs:1:1
  |
1 | / fn longest<'a>(x: &str, y: &str) -> &'a str {
2 | |     let result = String::from("really long string");
3 | |         result.as_str()
4 | |     }
  | | ^
```

The problem is that `result` goes out of scope and gets cleaned up at the end of the function. We're also trying to return a reference to `result` from the function, so we can't specify lifetime parameters that would change the dangling reference, as that would create a dangling reference. In this case, the best fix would be to return an owned copy of the string instead of 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 references within functions. Once they're connected, Rust has enough information to allow us to do useful things like return references and disallow operations that would create dangling pointers or otherwise violate the rules of memory safety.

Lifetime Annotations in Struct Definitions

So far, we've only defined structs to hold owned types. It's possible for struct fields to have different lifetimes than the struct itself. In that case we would need to add a lifetime annotation on every reference in the struct. Listing 10-25 has a struct named `ImportantExcerpt` that holds a string slice with a specific lifetime.

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. To make sure that the data in the struct doesn't outlive the references it holds, we declare the name of the generic lifetime parameter inside angle brackets (`'a`) after the struct so we can use the lifetime parameter in the body of the struct definition. This means an instance of `ImportantExcerpt` can't outlive the reference it holds.

The `main` function here creates an instance of the `ImportantExcerpt` struct and passes a reference to the first sentence of the `String` owned by the variable `novel`. The data in the `ImportantExcerpt` instance is created. In addition, `novel` doesn't go out of scope until `ImportantExcerpt` goes out of scope, so the reference in the `ImportantExcerpt` instance can't outlive the reference it holds.

Lifetime Elision

You've learned that every reference has a lifetime and that you need to specify lifetime annotations for functions or structs that use references. However, in Chapter 4 we had a function defined in Listing 4-9 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, though the parameter and return type are references

The reason this function compiles without lifetime annotations is historical: in the early days (pre-1.0) of Rust, this code wouldn't have compiled because every reference had to have a lifetime annotation. 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 using lifetime annotations over and over in particular situations. These situations 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.

need explicit annotations.

This piece of Rust history is relevant because it's possible that more determine and be added to the compiler. In the future, even fewer lifetime annotations will be required.

The patterns programmed into Rust's analysis of references are called the *lij*. They aren't rules for programmers to follow; they're a set of particular cases that the compiler can consider, and if your code fits these cases, you don't need to write the lifetime annotations.

The elision rules don't provide full inference. If Rust deterministically applies them, it will still ambiguity as to what lifetimes the references have, the compiler won't guess the remaining references should be. In this case, instead of guessing, the compiler will error that you can resolve by adding the lifetime annotations that specify how the references relate to each other.

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 what lifetimes references have without 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 annotations left, the compiler will stop with an error.

The first rule is that each parameter that is a reference gets its own lifetime parameter. 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 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 if there are multiple input lifetime parameters, but one of them is `&mut self` because this is a method, the lifetime of `self` is assigned to all other parameters. This third rule makes methods much nicer to read and write because it's necessary.

Let's pretend we're the compiler. We'll apply these rules to figure out what the references in the signature of the `first_word` function in Listing 10-26 are. The function signature is:

```
fn first_word(s: &str) -> &str {
```

Then the compiler applies the first rule, which specifies that each parameter has its own lifetime. We'll call it `'a` as usual, so now the signature is this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

The second rule applies because there is exactly one input lifetime. The second rule says that the lifetime of the one input parameter gets assigned to the output lifetime, so the final signature is this:

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

Now all the references in this function signature have lifetimes, and the compiler can analyze the function without needing the programmer to annotate the lifetimes in this function.

Let's look at another example, this time using the `longest` function that has been part of the standard library since 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 lifetimes 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 reference to `x`. The third rule doesn't apply either, because `longest` is a function rather than a method. Both 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. The compiler worked through the lifetime elision rules but still couldn't figure out what references in the signature.

Because the third rule really only applies in method signatures, we'll look at methods next to see why the third rule means we don't have to annotate lifetimes in method signatures often.

Lifetime Annotations in Method Definitions

When we implement methods on a struct with lifetimes, we use the same syntax as the type parameters shown in Listing 10-11. Where we declare and use the lifetime annotations 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` block and before 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 struct's fields, or they might be independent. In addition, the lifetime elision rules mean that lifetime annotations aren't necessary in method signatures. Let's look at an example 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 use after the type name is not required to annotate the lifetime of the reference to `self` because the reference is independent of the field.

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 `self` their own lifetimes. Then, because one of the parameters is `self`, `self` 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 the entire program. All string literals have the `'static` lifetime, which we can annotate:

```
let s: &'static str = "I have a static lifetime.>";
```

The text of this string is stored directly in the binary of your program, which means it's available for the entire lifetime of the program. Therefore, the lifetime of all string literals is `'static`.

You might see suggestions to use the `'static` lifetime in error messages. But if you're using `'static` as the lifetime for a reference, think about whether the reference needs to be valid for the entire lifetime of your program or not. You might consider whether you even need to use `'static`. Most of the time, the problem results from attempting to create a mismatch of the available lifetimes. In such cases, the solution is fixing the code by 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 together in one function!

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, announcement: T)
    where T: Display
{
    println!("Announcement! {}", announcement);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This is the `longest` function from Listing 10-22 that returns the longer of two strings. It has an extra parameter named `ann` of the generic type `T`, which can be any type that implements the `Display` trait as specified by the `where` clause. This extra parameter is necessary because the function compares the lengths of the string slices, which is why the `len()` method needs to be available. Because lifetimes are a type of generic, the declarations of the generic type parameters `x` and `y` 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, trait bounds, and generic lifetime parameters, you're ready to write code without being limited by many different situations. Generic type parameters let you apply the code to multiple types, and trait bounds ensure that even though the types are generic, they'll have specific needs. You learned how to use lifetime annotations to ensure that this flexibility doesn't lead to dangling references. And all of this analysis happens at compile time, which makes Rust a safe language.

performance!

Believe it or not, there is much more to learn on the topics we discussed in this chapter. Chapter 18 covers trait objects, which are another way to use traits. Chapter 19 covers scenarios involving lifetime annotations as well as some advanced type system features. In this chapter, you'll learn how to write tests in Rust so you can make sure your code is working correctly.

Writing Automated Tests

In his 1972 essay "The Humble Programmer," Edsger W. Dijkstra said that "Programmers should not just write programs; they should prove them correct." This is a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing the absence of bugs. 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 intended. We can design our programs with a high degree of concern about the correctness of programs, but they can be complex and not easy to prove. Rust's type system shoulders a huge part of the responsibility for correctness. While Rust's type system cannot catch every kind of incorrectness, it includes automated software tests within the language.

As an example, say we write a function called `add_two` that adds 2 to whatever it is passed. This function's signature accepts an integer as a parameter and returns an integer. When we implement and compile that function, Rust does all the type checking for us. You've learned so far to ensure that, for instance, we aren't passing a `String` to a function that expects an `int`. Rust can't check that this function will do precisely what we expect it to do, though. It's possible that `add_two(3)` will return the parameter plus 2 rather than, say, the parameter plus 10 or 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 the code to make sure that any existing correct behavior has not changed.

Testing is a complex skill: although we can't cover every detail about how to write tests in this chapter, we'll discuss the mechanics of Rust's testing facilities. We'll talk about the `#[test]` attribute, the macros available to you when writing your tests, the default behavior and options 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 correctly. 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 care of these steps. First, every test function must have the `#[test]` attribute. This attribute includes 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. This attribute provides metadata about pieces of Rust code; one example is the `derive` attribute we covered in Chapter 5. To change a function into a test function, add `#[test]` on the line before the function definition. When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs all the test functions in your project.

functions annotated with the `test` attribute and reports on whether each test fails.

In Chapter 7, we saw that when we make a new library project with Cargo, a `tests` module in it is automatically generated for us. This module helps you start your tests quickly without having to look up the exact structure and syntax of test functions every time you add them. You can add as many additional test functions and as many test modules as you like.

We'll explore some aspects of how tests work by experimenting with the test module in our `adder` library without actually testing any code. Then we'll write some real-world tests that check that the `adder` function does what we've written and assert that its behavior is correct.

Let's create a new library project called `adder`:

```
$ 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() {
        assert_eq!(2 + 2, 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 to see how it works. The `#[test]` annotation before the `fn` line tells the test runner to treat this function as a test. We could also have non-test functions in the `tests` module to help set up common scenarios or perform common operations, such as creating temporary files. We can tell which functions are tests by using the `#[test]` attribute.

The function body uses the `assert_eq!` macro to assert that $2 + 2$ equals 4. This is 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 dev [unoptimized + debuginfo] target(s) in 0.22 secs
     Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

Listing 11-2: The output from running the automatically generated test

Cargo compiled and ran the test. After the `Compiling`, `Finished`, and `Running` messages, the test output shows that one test passed.

running 1 test. The next line shows the name of the generated test function and the result of running that test, ok. The overall summary of running the test test result: ok. means that all the tests passed, and the portion that 1 passed; 0 failed totals the number of tests that passed or failed.

Because we don't have any tests we've marked as ignored, the summary hasn't filtered the tests being run, so the end of the summary shows 0 failing. See the documentation about ignoring and filtering out tests in the next section, "Controlling How Tests Run".

The 0 measured statistic is for benchmark tests that measure performance. This feature is currently only available in nightly Rust. See the documentation about benchmarks for more.

The next part of the test output, which starts with doc-tests adder, is for documentation tests. We don't have any documentation tests yet, but Rust contains examples that appear in our API documentation. This feature helps us keep our documentation in sync! We'll discuss how to write documentation tests in the "Documentation" chapter of Chapter 14. For now, we'll ignore the doc-tests output.

Let's change the name of our test to see how that changes the test output. Change the function name 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 adder.

```
running 1 test
test tests::exploration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Let's add another test, but this time we'll make a test that fails! Tests fail when they panic. Each test is run in a new thread, and when the main thread has died, the test is marked as failed. We talked about the simplest way to cause a panic, which is to call the panic! macro. Enter the new test, 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: our `exploration` test passed and `another` failed:

```
running 2 tests
test tests::exploration ... ok
test tests::another ... FAILED

failures:

---- tests::another stdout ----
    thread 'tests::another' panicked at 'Make this test fail', src/
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::another

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 ·
error: test failed
```

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 show individual results and the summary: the first section displays the detailed results of each failing test. In this case, `another` failed because it `panicked` at 'Make this test fail'. The next section lists just the names of failing tests. This is useful when there are lots of tests and lots of failing tests. You can run just that test to more easily debug it; we'll talk more about how to do this in the "Controlling How Tests Are Run" section.

The summary line displays at the end: overall, our test result is `FAILED`. We know that one test failed.

Now that you've seen what the test results look like in different scenarios, let's look at other ways to write tests 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 check that a condition in a test evaluates to `true`. We give the `assert!` macro an argument that is a Boolean expression. If the value is `true`, `assert!` does nothing and the test passes. If the value is `false`, `assert!` calls the `panic!` macro, which causes the test to fail. Using `assert!` 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. Let's use this code here in Listing 11-5. Let's put this code in the `src/lib.rs` file and write some tests that use the `assert!` macro.

Filename: `src/lib.rs`

```

#[derive(Debug)]
pub struct Rectangle {
    length: u32,
    width: u32,
}

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

```

Listing 11-5: Using the `Rectangle` struct and its `can_hold` method from Chapter 11

The `can_hold` method returns a Boolean, which means it's a perfect use case for a macro. In Listing 11-6, we write a test that exercises the `can_hold` method by creating two `Rectangle` instances that has a length of 8 and a width of 7 and asserting that it can hold another instance that has a length of 5 and a width of 1:

Filename: `src/lib.rs`

```

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

    #[test]
    fn larger_can_hold_smaller() {
        let larger = Rectangle { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(larger.can_hold(&smaller));
    }
}

```

Listing 11-6: A test for `can_hold` that checks whether a larger rectangle can hold a smaller rectangle

Note that we've added a new line inside the `tests` module: `use super::*;`; this is a regular module that follows the usual visibility rules we covered in Chapter 7. Because the `tests` module is an inner module, we need to bring the outer module into the scope of the inner module. We use a glob here so any code 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 we expect the test to pass. Let's find out!

```

running 1 test
test tests::larger_can_hold_smaller ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

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 { length: 8, width: 7 };
        let smaller = Rectangle { length: 5, width: 1 };

        assert!(!smaller.can_hold(&larger));
    }
}

```

Because the correct result of the `can_hold` function in this case is `false`, we get a `assert!` result before we pass it to the `assert!` macro. As a result, our test will pass as `false`:

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

```

Two tests that pass! Now let's see what happens to our test results when we change the implementation of the `can_hold` method by replacing the greater-than sign with a less-than sign when it compares the lengths:

```

// --snip--

impl Rectangle {
    pub fn can_hold(&self, other: &Rectangle) -> bool {
        self.length < other.length && self.width > other.width
    }
}

```

Running the tests now produces the following:

```

running 2 tests
test tests::smaller_cannot_hold_larger ... ok
test tests::larger_can_hold_smaller ... FAILED

failures:
---- tests::larger_can_hold_smaller stdout ----
    thread 'tests::larger_can_hold_smaller' panicked at 'assertion failed: `(left == right)`'
      larger.can_hold(&smaller)', src/lib.rs:22:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::larger_can_hold_smaller

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered

```

Our tests caught the bug! Because `larger.length` is 8 and `smaller.length` is 5, the `can_hold` method now returns `false`: 8 is not less than 5.

Testing Equality with the `assert_eq!` and `assert_ne!` Macros

A common way to test functionality is to compare the result of the code under test with what you expect it to return to make sure they're equal. You could do this using the `==` operator, but this is such a standard library provides a pair of macros—`assert_eq!` and `assert_ne!`—more conveniently. These macros compare two arguments for equality or inequality. They'll also print the two values if the assertion fails, which makes it easier to debug. Conversely, the `assert!` macro only indicates that it got a `false` value for the condition, not the values that lead 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!

```
running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

The first argument we gave to the `assert_eq!` macro, `4`, is equal to the result of `add_two(2)`. The line for this test is `test tests::it_adds_two ... ok`, and that means our test passed!

Let's introduce a bug into our code to see what it looks like when a test 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:

```
running 1 test
test tests::it_adds_two ... FAILED

failures:

---- tests::it_adds_two stdout ----
     thread 'tests::it_adds_two' panicked at 'assertion failed:
      left: `4`,
      right: `5`, src/lib.rs:11:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::it_adds_two

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 ·
```

Our test caught the bug! The `it_adds_two` test failed, displaying the message `assertion failed: `(left == right)`` and showing that `left` was `4` and `right` was `5`. This message is useful and helps us start debugging: it means the `left` argument had the value we expected, but the `right` argument, where we had `add_two(2)`, was `5`.

Note that in some languages and test frameworks, the parameters to the function under test are called `expected` and `actual`, and the order in which they're passed doesn't matter. In Rust, however, they're called `left` and `right`, and the order in which they're passed does matter: `assert_eq!(left, right)` checks that `left` has the value we expect and the value that the code under test produces doesn't matter. If we wanted to assert that `left` was `5` and `right` was `4`, we would do `assert_eq!(right, left)`.

The `assert_ne!` macro will pass if the two values we give it are not equal and fail if they are. This macro is most useful for cases when we're not sure what a value *will* be or *won't* be if our code is functioning as we intend. For example, consider a function that is guaranteed to change its input in some way, but the way in which it changes depends on the day of the week that we run our tests, the best thing to do is to assert that the output of the function is not equal to the input.

Under the surface, the `assert_eq!` and `assert_ne!` macros use the `PartialEq` and `Debug` traits respectively. When the assertions fail, these macros print their arguments using the `Debug` trait, which means the values being compared must implement the `PartialEq` and `Debug` traits. Most primitive types and most of the standard library types implement these traits, so you can use them with most values. If you define your own type, you'll need to implement `PartialEq` to assert that values of that type are equal or not equal. You'll need to implement `Debug` to print the values when the assertion fails. Both traits are derivable traits, as mentioned in Listing 5-12 in Chapter 5, so it's as straightforward as adding the `#[derive(PartialEq, Debug)]` annotation to your definition. See Appendix C, "Derivable Traits," for more details about these traits.

Adding Custom Failure Messages

You can also add a custom message to be printed with the failure message for the `assert!`, `assert_eq!`, and `assert_ne!` macros. Any arguments specified after the required argument to `assert!` or the two required arguments to `assert_eq!` or `assert_ne!` are passed along to the `format!` macro (discussed in Chapter 8 in the "Formatting Output with the `format!` Macro" section), so you can pass a format string that includes placeholders and values to go in those placeholders. Custom messages are useful because they tell you exactly what an assertion means; when a test fails, you'll have a better idea of what went wrong.

For example, let's say we have a function that greets people by name and we want to make sure 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 free to change them. For example, the text at the beginning of the greeting will change. We decided we don't want to change the code in the function itself, so instead of checking for exact equality, we'll just assert that the output contains the text "Hello". This way, if the requirements change, we can change the test when the requirements change, so instead of checking for exact equality, we'll just assert that the output contains the text "Hello".

Let's introduce a bug into this code by changing `greeting` to not include `Hello`. The failure looks like:

```
pub fn greeting(name: &str) -> String {
    String::from("Hello!")
}
```

Running this test produces the following:

```
running 1 test
test tests::greeting_contains_name ... FAILED

failures:

---- tests::greeting_contains_name stdout ----
      thread 'tests::greeting_contains_name' panicked at 'assertion failed: `result.contains("Carol")`'
      result: `result.contains("Carol")`, src/lib.rs:12:8
      note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::greeting_contains_name
```

This result just indicates that the assertion failed and which line the assertion failed on. A more useful failure message in this case would print the value we got from the `greeting` function, giving it a custom failure message made from a format string 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 was {}",
        result,
        result
    );
}
```

Now when we run the test, we'll get a more informative error message:

```
---- tests::greeting_contains_name stdout ----
     thread 'tests::greeting_contains_name' panicked at 'Greeti
contain name, value was `Hello!`', src/lib.rs:12:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

We can see the value we actually got in the test output, which would help us instead of what we were expecting to happen.

Checking for Panics with `should_panic`

In addition to checking that our code returns the correct values we expect, it check that our code handles error conditions as we expect. For example, cor that we created in Chapter 9, Listing 9-9. Other code that uses `Guess` depen that `Guess` instances will contain only values between 1 and 100. We can wr that attempting to create a `Guess` instance with a value outside that range p

We do this by adding another attribute, `should_panic`, to our test function. test pass if the code inside the function panics; the test will fail if the code in panic.

Listing 11-8 shows a test that checks that the error conditions of `Guess::new` expect them to:

Filename: `src/lib.rs`

```
pub struct Guess {
    value: u32,
}

impl Guess {
    pub fn new(value: u32) -> 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 be applies to. Let's look at the result when this test passes:

```
running 1 test
test tests::greater_than_100 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Looks good! Now let's introduce a bug in our code by removing the condition that will panic if the value is greater than 100:

```
// --snip--  
  
impl Guess {  
    pub fn new(value: u32) -> 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:

```
running 1 test  
test tests::greater_than_100 ... FAILED  
  
failures:  
  
failures:  
    tests::greater_than_100  
  
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 .
```

We don't get a very helpful message in this case, but when we look at the test, it's annotated with `#[should_panic]`. The failure we got means that the code did not cause a panic.

Tests that use `should_panic` can be imprecise because they only indicate that some panic happened. A `should_panic` test would pass even if the test panics for a different reason than the one we were expecting to happen. To make `should_panic` tests more precise, add an optional `expected` parameter to the `should_panic` attribute. The test harness will then compare the failure message to the provided text. For example, consider the modified code 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: u32) -> Guess {  
        if value < 1 {  
            panic!("Guess value must be greater than or equal to 1,  
                   value);  
        } else if value > 100 {  
            panic!("Guess value must be less than or equal to 100,  
                   value);  
        }  
  
        Guess {  
            value  
        }  
    }  
}  
  
#[cfg(test)]  
mod tests {  
    use super::*;

    #[test]  
    #[should_panic(expected = "Guess value must be less than or equal to 100, got 200")]
    fn greater_than_100() {  
        Guess::new(200);  
    }
}
```

Listing 11-9: Testing that a condition will cause a `panic!` with a particular panic message

This test will pass because the value we put in the `should_panic` attribute's `expected` substring of the message that the `Guess::new` function panics with. We can verify that 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 change here depends on how much of the panic message you care about. The `expected` parameter for `should_panic` depends on how much of the panic message you care about. In this case, a substring of the message is enough to ensure that the code in the test function executes the `else if` block.

To see what happens when a `should_panic` test with an `expected` message fails, let's introduce a bug into our code by swapping the bodies of the `if value < 1` and `else if value > 100` blocks:

```
if value < 1 {  
    panic!("Guess value must be less than or equal to 100, got {}.");  
} else if value > 100 {  
    panic!("Guess value must be greater than or equal to 1, got {}.");  
}
```

This time when we run the `should_panic` test, it will fail:

```
running 1 test
test tests::greater_than_100 ... FAILED

failures:

---- tests::greater_than_100 stdout ----
    thread 'tests::greater_than_100' panicked at 'Guess value must be greater than or equal to 1, got 200.', src/lib.rs:11:12
note: Run with `RUST_BACKTRACE=1` for a backtrace.
note: Panic did not include expected string '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 .
```

The failure message indicates that this test did indeed panic as we expected. It did not include the expected string 'Guess value must be less than or equal to 100'. The message that we did get in this case was
'Guess value must be greater than or equal to 1, got 200.' Now we can see where our bug is!

Now that you know several ways to write tests, let's look at what is happening behind the scenes 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. You can specify command line options to change the default behavior of `cargo test`. For example, the default behavior produced by `cargo test` is to run all the tests in parallel and capture output from each test separately. If you want to run the tests sequentially, you can use the `--parallel` option. If you want to capture the output of all tests together, you can use the `--nocapture` option. If you want to run the tests in parallel but capture the output of all tests together, you can use the `--parallel` option with the `--nocapture` option.

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` before the separator `--` and then the ones that go to the test binary. Running `cargo test -- --help` shows the options you can use with `cargo test`, and running `cargo test -- --help` shows 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. This allows them to finish running faster so you can get feedback quicker on whether or not your tests pass. However, because the tests are running at the same time, make sure your tests don't depend 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 and writes some data to that file. Then each test reads the data in that file and checks if it contains a particular value, which is different in each test. Because the tests run in parallel, one test might overwrite the file between when another test writes and reads it. This will then fail, not because the code is incorrect but because the tests have different results 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 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 parallel threads. Running tests using one thread will take longer than running them in parallel, but the tests will not 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 output in the terminal; we'll see only the line that indicates the test passed. If a test fails, we'll see the output 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 `a` and two tests that check the value it returns, one 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:

```

running 2 tests
test tests::this_test_will_pass ... ok
test tests::this_test_will_fail ... FAILED

failures:

---- tests::this_test_will_fail stdout ----
    I got the value 8
thread 'tests::this_test_will_fail' panicked at 'assertion failed:
  left: `5`,
  right: `10`, src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 ·

```

Note that nowhere in this output do we see `I got the value 4`, which is what the test that passes runs. That output has been captured. The output from the test that fails, `I got the value 8`, appears in the section of the test summary output, which is part of the test failure.

If we want to see printed values for passing tests as well, we can disable the capturing by using the `--nocapture` flag:

```
$ cargo test -- --nocapture
```

When we run the tests in Listing 11-10 again with the `--nocapture` flag, we get:

```

running 2 tests
I got the value 4
I got the value 8
test tests::this_test_will_pass ... ok
thread 'tests::this_test_will_fail' panicked at 'assertion failed:
  left: `5`,
  right: `10`, src/lib.rs:19:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.
test tests::this_test_will_fail ... FAILED

failures:

failures:
    tests::this_test_will_fail

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 ·

```

Note that the output for the tests and the test results are interleaved; the results are running in parallel, as we talked about in the previous section. Try using the `--no-run-processes` option and the `--nocapture` flag, and see what the output looks like then!

Running a Subset of Tests by Name

Sometimes, running a full test suite can take a long time. If you're working on a specific area, you might want to run only the tests pertaining to that code. You can do this by passing `cargo test` the name or names of the test(s) you want to run as arguments.

To demonstrate how to run a subset of tests, we'll create three tests for our `lib.rs` file, 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

```

running 3 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok
test tests::one_hundred ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

```

Running Single Tests

We can pass the name of any test function to `cargo test` to run only that test:

```

$ cargo test one_hundred
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running target/debug/deps/adder-06a75b4a1f2515e9

running 1 test
test tests::one_hundred ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered

```

Only the test with the name `one_hundred` ran; the other two tests didn't match. The output lets us know we had more tests than what this command ran by displaying the count at the end of the summary line.

We can't specify the names of multiple tests in this way; only the first value given 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 part will run. For example, because two of our tests' names contain `add`, we can run those two tests by running `cargo test add`:

```
$ cargo test add
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/adder-06a75b4a1f2515e9

running 2 tests
test tests::add_two_and_two ... ok
test tests::add_three_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

This command ran all tests with `add` in the name and filtered out the test named `expensive_test`. Note that the module in which tests appear becomes part of the test's name. You can filter 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 skip them during most runs of `cargo test`. Rather than listing as arguments all the tests you want to skip, you can instead annotate the time-consuming tests using the `ignore` attribute, 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 `cargo test`, `it_works` runs, but `expensive_test` doesn't:

```
$ cargo test
    Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24 secs
    Running target/debug/deps/adder-ce99bcc2479f4607

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
```

The `expensive_test` function is listed as `ignored`. If we want to run only the `expensive_test` test, we can use `cargo test -- --ignored`:

```
$ cargo test -- --ignored
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/adder-ce99bcc2479f4607

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out
```

By controlling which tests run, you can make sure your `cargo test` results are at a point where it makes sense to check the results of the `ignored` tests and for the results, you can run `cargo test -- --ignored` instead.

Test Organization

As mentioned at the start of the chapter, testing is a complex discipline, and different terminology and organization. The Rust community thinks about tests in categories: *unit tests* and *integration tests*. Unit tests are small and more focused on isolation at a time, and can test private interfaces. Integration tests are end-to-end tests of your library and use your code in the same way any other external code would, using its public interface and potentially exercising multiple modules per test.

Writing both kinds of tests is important to ensure that the pieces of your library work together and that they expect them to be used separately and together.

Unit Tests

The purpose of unit tests is to test each unit of code in isolation from the rest of the system, so that you can pinpoint exactly where code is failing and isn't working as expected. You'll put unit tests in the same files as the code that they're testing. The convention is to create a module named `tests` 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 link the module only when you run `cargo test`, not when you run `cargo build`. This saves compilation time when you want to build the library and saves space in the resulting compiled artifact because the tests won't be included. You'll see that because integration tests go in a different directory, they don't have the `#[cfg(test)]` annotation. However, because unit tests go in the same files as the code they're testing, we need 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, we generated this code for us:

Filename: `src/lib.rs`

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

This code is the automatically generated test module. The attribute `cfg(test)` tells Rust that the following item should only be included given a certain configuration. In this case, the configuration option is `test`, which is provided by Rust for compilation. When we run `cargo test`, Cargo compiles our test code only if we actively run `cargo test`. This includes any helper functions that might be within this module, such as the `it_works` function 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 a good job of supporting both approaches.

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`, but because it's part of the same module as the test code, and the `tests` module is just another module, you can import and call `internal_adder` from the `tests` module just fine. If you don't think private functions should be tested, there's nothing stopping you from marking them as `pub`.

Integration Tests

In Rust, integration tests are entirely external to your library. They use your library as if it were any other code would, which means they can only call functions that are part of the public API. Their purpose is to test whether many parts of your library work together correctly. Functions that work correctly on their own could have problems when integrated with other parts of the system. Testing the integrated code is important as well. To create integration tests, you first need to create a `tests` directory.

The `tests` Directory

We create a `tests` directory at the top level of our project directory, next to `src`. We can then put all of our integration test files in this directory. We can then make as many test files as we want in this directory, 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, create a new file named `tests/integration_test.rs`, and enter the code below.

Filename: tests/integration_test.rs

```
extern crate 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

We've added `extern crate adder` at the top of the code, which we didn't need before. The reason is that each test in the `tests` directory is a separate crate, so we need to bring in the `adder` crate into each of them.

We don't need to annotate any code in `tests/integration_test.rs` with `#[cfg(test)]`. The `cargo test` command will automatically run tests in the `tests` directory specially and compiles files in this directory only when we run `cargo test` now:

```
$ cargo test
    Compiling adder v0.1.0 (file:///projects/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
        Running target/debug/deps/adder-abcabcabc

running 1 test
test tests::internal ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

    Running target/debug/deps/integration_test-ce99bcc2479f4607

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

The three sections of output include the unit tests, the integration test, and the documentation tests. The first section for the unit tests is the same as we've been seeing: one line for each `internal` function 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 target/debug/deps/integration-test-ce99bcc2479f4607` (the hash in the output will be different). Next, there is a line for each test function in that integration test file, followed by a summary line for the results of the integration test just before the `Doc-tests` section.

Similarly to how adding more unit test functions adds more result lines to the unit tests section, adding more test functions to the integration test file adds more result lines to the integration tests section. Each integration test file has its own section, so if we add more integration test files to the `tests` directory, there will be more integration test sections.

We can still run a particular integration test function by specifying the test function name as an argument to `cargo test`. To run all the tests in a particular integration test file, we can use the `--test` argument of `cargo test` followed by the name of the file:

```
$ cargo test --test integration_test
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running target/debug/integration_test-952a27e0126bb565

running 1 test
test it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

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 than one file to help organize them; for example, you can group the test functions by the same module. As mentioned earlier, each file in the `tests` directory is compiled as its own crate.

Treating each integration test file as its own crate is useful to create separate crates like the way end users will be using your crate. However, this means files in the `tests` directory share the same behavior as files in `src` do, as you learned in Chapter 7 regarding how to turn code into modules and files.

The different behavior of files in the `tests` directory is most noticeable when you have multiple test files that contain shared code. You could move those shared functions that would be useful in multiple integration test files and you try to move them into a separate file. In the “Moving Modules to Other Files” section of Chapter 7 to extract them into a separate file. For example, if we create `tests/common.rs` and place a function named `setup` in it, we can move the shared code into `common.rs` and then import it into each test file. In the code to `common.rs` 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` file, though this file doesn't contain any test functions nor did we call the `setup` function anywhere:

```
running 1 test  
test tests::internal ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Running target/debug/deps/common-b8b07b6f1be2db70  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Running target/debug/deps/integration_test-d993c68b431d39df  
  
running 1 test  
test it_adds_two ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out  
  
Doc-tests adder  
  
running 0 tests  
  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Having `common` appear in the test results with `running 0 tests` displayed for the `common` file 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/mod.rs`, we can create a `tests/common.rs`. In the “Rules of Module Filesystems” section of Chapter 7, we learned about the convention `module_name/mod.rs` for files of modules that have submodules. We can use this convention here, but naming the file this way tells Rust not to treat it as an integration test file. When we move the `setup` function code into `tests/common.rs`, we can delete the `tests/common.rs` file, and the section in the test output will no longer appear. Note that the subdirectories of the `tests` directory don't get compiled as separate crates or appear in the test output.

After we've created `tests/common/mod.rs`, we can use it from any of the integrations tests. Here's an example of calling the `setup` function from the `it_adds_two.rs`:

Filename: `tests/integration_test.rs`

```
extern crate 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 in Listing 7-4. 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 `tests` directory, we can't create integration tests in the `tests` directory and use `extern crate` to refer to the code in the `src/main.rs` file. Only library crates expose functions that other crates can use, so we can't use `extern crate` to refer to the code in the `src/main.rs` file. Instead, we have to define the `setup` function in the `src/main.rs` file and then call it from the `tests/integration_test.rs` file.

This is one of the reasons Rust projects that provide a binary have a straight `src/main.rs` file instead of a `src/lib.rs` file. Using that structure, integration tests can be run on their own. If the `src/main.rs` file has a `main` function, the test will work, and if it doesn't, the small amount of code in the `src/main.rs` file will work as well, and the test will still pass.

Summary

Rust's testing features provide a way to specify how code should function and verify that it does. Unit tests exercise different parts of your code separately and can test private implementation details. Integration tests check that different parts of your code work together correctly, and they use the library's public API to test how external code will use it. Even though Rust's type system and ownership rules can catch some kinds of bugs, tests are still important to reduce logic bugs having to do with how the code is expected to behave.

Let's combine the knowledge you learned in this chapter and in previous chapters to build a command-line tool for a real-world project!

An I/O Project: Building a Command Line Tool

This chapter is a recap of the many skills you've learned so far and an exploration of Rust's standard library features. We'll build a command line tool that interacts with the user via input/output to practice some of the Rust concepts you now have under your belt.

Rust's speed, safety, single binary output, and cross-platform support make it a great choice for creating command line tools, so for our project, we'll make our own version of `curl`.

line tool `grep` (globally search a regular expression and `print`). In the simple searches a specified file for a specified string. To do so, `grep` takes as its argument a string. Then it reads the file, finds lines in that file that contain the string and those lines.

Along the way, we'll show how to make our command line tool use features many command line tools use. We'll read the value of an environment variable to configure the behavior of our tool. We'll also print to the standard error console instead of standard output (`stdout`), so, for example, the user can redirect the file while still seeing error messages onscreen.

One Rust community member, Andrew Gallant, has already created a fully functional version of `grep`, called `ripgrep`. By comparison, our version of `grep` will be fairly simple, but it will give you some of the background knowledge you need to understand a more complex version like `ripgrep`.

Our `grep` project will combine a number of concepts you've learned so far:

- Organizing code (using what you learned in modules, Chapter 7)
- Using vectors and strings (collections, Chapter 8)
- Handling errors (Chapter 9)
- Using traits and lifetimes where appropriate (Chapter 10)
- Writing tests (Chapter 11)

We'll also briefly introduce closures, iterators, and trait objects, which Chapter 13 covers in detail.

Accepting Command Line Arguments

Let's create a new project with, as always, `cargo new`. We'll call our project `minigrep`, just like the `grep` tool that you might already have on your system.

```
$ cargo new minigrep
     Created binary (application) `minigrep` project
$ cd minigrep
```

The first task is to make `minigrep` accept its two command line arguments: a string to search for. That is, we want to be able to run our program with `cargo run searchstring filename.txt`, where `searchstring` is the string to search for, and `filename` is a path to a file to search in, like so:

```
$ cargo run searchstring example-filename.txt
```

Right now, the program generated by `cargo new` cannot process arguments. Existing libraries on [Crates.io](#) can help with writing a program that accepts command line arguments, but because you're just learning this concept, let's implement this ourselves.

Reading the Argument Values

To enable `minigrep` to read the values of command line arguments we pass the function provided in Rust's standard library, which is `std::env::args`. This function returns an iterator of the command line arguments that were given to `minigrep`. We haven't talked about iterators yet (we'll cover them fully in Chapter 13), but for now, you only need to know about iterators: iterators produce a series of values, and we can call the `collect` method on an iterator to turn it into a collection, such as a vector, containing all the elements.

produces.

Use the code in Listing 12-1 to allow your `minigrep` program to read any command-line arguments passed to it and then collect the values into a vector:

Filename: `src/main.rs`

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    println!("{}:", args);
}
```

Listing 12-1: Collecting the command line arguments into a vector and printing them

First, we bring the `std::env` module into scope with a `use` statement so we can call its `args` function. Notice that the `std::env::args` function is nested in two levels of scopes: `env` and `std`. This is discussed in Chapter 7, in cases where the desired function is nested in more than one module. It’s conventional to bring the parent module into scope rather than the function itself, so we bring `std` into scope here. We can then easily use other functions from `std::env`. It’s also less ambiguous than adding `use std::env::args` and then calling the function with just `args`, because `args` could be mistaken for a function that’s defined in the current module.

The `args` Function and Invalid Unicode

Note that `std::env::args` will panic if any argument contains invalid Unicode. If you need to accept arguments containing invalid Unicode, use `std::env::args_os`. This function returns an iterator that produces `OsString` values instead of `String` values. We chose to use `std::env::args` here for simplicity, because `OsString` values are platform-specific and are more complex to work with than `String` values.

On the first line of `main`, we call `env::args`, and we immediately use `collect` to turn the iterator into a vector containing all the values produced by the iterator. We can use `args` to create many kinds of collections, so we explicitly annotate the type of `args` as a `Vec<String>` vector of strings. Although we very rarely need to annotate types in Rust, doing so is often necessary because Rust isn’t able to infer the kind of collection you want to create.

Finally, we print the vector using the debug formatter, `{:?}`. Let’s try running `minigrep` with no arguments and then with two arguments:

```
$ cargo run
--snip--
["target/debug/minigrep"]

$ cargo run needle haystack
--snip--
["target/debug/minigrep", "needle", "haystack"]
```

Notice that the first value in the vector is `"target/debug/minigrep"`, which is the path to the binary. This matches the behavior of the arguments list in C, letting programs know which binary they were invoked in during their execution. It’s often convenient to have access to the name of the binary in case you want to print it in messages or change the behavior of the program. In this case, a command-line alias was used to invoke the program. But for the purposes of this example, we’ll ignore it and save only the two arguments we need.

Saving the Argument Values in Variables

Printing the value of the vector of arguments illustrated that the program is specified as command line arguments. Now we need to save the values of the variables so we can use the values throughout the rest of the program. We can do this by creating variables:

Filename: src/main.rs

```
use std::env;

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

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

Listing 12-2: Creating variables to hold the query argument and filename arguments

As we saw when we printed the vector, the program's name takes up the first element, `args[0]`, so we're starting at index `1`. The first argument `minigrep` takes is the query, so we put a reference to the first argument in the variable `query`. The second argument will be the filename, so we put a reference to the second argument in the variable `filename`.

We temporarily print the values of these variables to prove that the code is working correctly. Let's run this program again with the arguments `test` and `sample.txt`:

```
$ cargo run test sample.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/minigrep test sample.txt'
Searching for test
In file sample.txt
```

Great, the program is working! The values of the arguments we need are being stored in variables. Later we'll add some error handling to deal with certain potential errors, such as when the user provides no arguments; for now, we'll ignore that situation and add file-reading capabilities instead.

Reading a File

Now we'll add functionality to read the file that is specified in the `filename` argument. First, we need a sample file to test it with: the best kind of file to use for testing if `minigrep` is working is one with a small amount of text over multiple lines with punctuation and words. Listing 12-3 has an Emily Dickinson poem that will work well! Create a new file named `poem.txt` at the root level of your project, and enter the poem "I'm Nobody! Who are you?"

Filename: poem.txt

I'm nobody! Who are you?
 Are you nobody, too?
 Then there's a pair of us – don't tell!
 They'd banish us, you know.

How dreary to be somebody!
 How public, like a frog
 To tell your name the livelong day
 To an admiring bog!

Listing 12-3: A poem by Emily Dickinson makes a good test case

With the text in place, edit `src/main.rs` and add code to open the file, as shownFilename: `src/main.rs`

```
use std::env;
use std::fs;
use std::io::prelude::*;

fn main() {
    // --snip--
    println!("In file {}", filename);

    let contents = fs::read_to_string(filename)
        .expect("Something went wrong reading the file");

    println!("With text:\n{}", contents);
}
```

Listing 12-4: Reading the contents of the file specified by the second argument

First, we add some more `use` statements to bring in relevant parts of the `std::fs` to handle files, and `std::io::prelude::*` contains various useful including file I/O. In the same way that Rust has a general prelude that brings functions into scope automatically, the `std::io` module has its own prelude functions you'll need when working with I/O. Unlike with the default prelude a `use` statement for the prelude from `std::io`.

In `main`, we've added a new statement: `fs::read_to_string` will take the `filename` and then produce a new `String` with its contents.

After those lines, we've again added a temporary `println!` statement that prints `contents` after the file is read, so we can check that the program is working.

Let's run this code with any string as the first command line argument (because we haven't implemented the searching part yet) and the `poem.txt` file as the second argument:

```
$ cargo run the poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
  Running `target/debug/minigrep the poem.txt`
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us – don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

Great! The code read and then printed the contents of the file. But the code `main` function has multiple responsibilities: generally, functions are clearer if each function is responsible for only one idea. The other problem is that we've written more code than we need to. As our program grows, it will be harder to fix them cleanly. It's good practice to begin refactoring as soon as you start developing a program, because it's much easier to refactor smaller amounts of code at a time than to go back and fix a large amount of code later.

Refactoring to Improve Modularity and Error Handling

To improve our program, we'll fix four problems that have to do with the `main` function: how it's performing multiple tasks, how it's handling potential errors, how it's combining configuration variables, and how it's handling user input.

First, our `main` function now performs two tasks: it parses arguments and opens a file. While this is fine for a small function, this isn't a major problem. However, if we continue to grow our program, the number of separate tasks the `main` function handles will increase. As a program grows, it becomes more difficult to reason about, harder to test, and harder to maintain without breaking one of its parts. It's best to separate functionality so each function is responsible for one task.

This issue also ties into the second problem: although `query` and `filename` are passed as arguments to our program, variables like `f` and `contents` are used to perform the work. The longer `main` becomes, the more variables we'll need to bring into scope. As a program grows, having many variables in scope, the harder it will be to keep track of the purpose of each. It's better to group configuration variables into one structure to make their purpose clear.

The third problem is that we've used `expect` to print an error message when a file is missing, but the error message just prints `file not found`. Opening a file can fail for reasons besides the file being missing: for example, the file might exist, but we might not have permission to open it. Right now, if we're in that situation, we'd print the `file not found` message, which would give the user the wrong information!

Fourth, we use `expect` repeatedly to handle different errors, and if the user provides too few or too many arguments, they'll get an `index out of bounds` error. This doesn't clearly explain the problem. It would be best if all the error-handling code was in one place, so future maintainers had only one place to consult in the code if the error-handling logic changes. Having all the error-handling code in one place will also ensure that the error messages are meaningful to our end users.

Let's address these four problems by refactoring our project.

Separation of Concerns for Binary Projects

The organizational problem of allocating responsibility for multiple tasks to a single function is common to many binary projects. As a result, the Rust community has developed a guideline for splitting the separate concerns of a binary program when it becomes large. The process has the following steps:

- Split your program into a `main.rs` and a `lib.rs` and move your program's configuration logic to `lib.rs`.
- As long as your command line parsing logic is small, it can remain in `main.rs`.
- When the command line parsing logic starts getting complicated, extra logic or shared state between `main.rs` and `lib.rs`, move it to `lib.rs`.

- The responsibilities that remain in the `main` function after this process following:
 - Calling the command line parsing logic with the argument values
 - Setting up any other configuration
 - Calling a `run` function in `lib.rs`
 - Handling the error if `run` returns an error

This pattern is about separating concerns: `main.rs` handles running the program all the logic of the task at hand. Because you can't test the `main` function directly, you test all of your program's logic by moving it into functions in `lib.rs`. The original `main.rs` will be small enough to verify its correctness by reading it. Let's rework following this process.

Extracting the Argument Parser

We'll extract the functionality for parsing arguments into a function that moves the command line parsing logic to `src/lib.rs`. Listing 12-5 shows the new code that calls a new function `parse_config`, which we'll define in `src/main.rs` for the rest of this chapter.

Filename: `src/main.rs`

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

    let (query, filename) = parse_config(&args);

    // --snip--
}

fn parse_config(args: &[String]) -> (&str, &str) {
    let query = &args[1];
    let filename = &args[2];

    (query, filename)
}
```

Listing 12-5: Extracting a `parse_config` function from `main`

We're still collecting the command line arguments into a vector, but instead of passing each argument value at index `1` to the variable `query` and the argument value at index `2` to the variable `filename` within the `main` function, we pass the whole vector to the `parse_config` function. This function then holds the logic that determines which argument corresponds to which variable and passes the values back to `main`. We still create the `query` and `filename` variables in `main`, but `main` no longer has the responsibility of determining how the command line arguments correspond to the variables.

This rework may seem like overkill for our small program, but we're refactoring to make the code more modular. After making this change, run the program again to verify that the argument parser works. It's good to check your progress often, to help identify the cause of potential bugs before they occur.

Grouping Configuration Values

We can take another small step to improve the `parse_config` function further. Instead of returning a tuple, we're returning a tuple, but then we immediately break that tuple into individual variables. This is a sign that perhaps we don't have the right abstraction yet.

Another indicator that shows there's room for improvement is the `config` parameter.

which implies that the two values we return are related and are both part of We're not currently conveying this meaning in the structure of the data other two values into a tuple; we could put the two values into one struct and give a meaningful name. Doing so will make it easier for future maintainers of this how the different values relate to each other and what their purpose is.

Note: Some people call this anti-pattern of using primitive values when a more appropriate *primitive obsession*.

Listing 12-6 shows the addition of a struct named `Config` defined to have fields `filename`. We've also changed the `parse_config` function to return an instance of the struct and updated `main` to use the struct fields rather than having separate variables.

Filename: src/main.rs

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

    let config = parse_config(&args);

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    let contents = fs::read_to_string(config.filename)
        .expect("Something went wrong reading the file");

    // --snip--
}

struct Config {
    query: String,
    filename: String,
}

fn parse_config(args: &[String]) -> Config {
    let query = args[1].clone();
    let filename = args[2].clone();

    Config { query, filename }
}
```

Listing 12-6: Refactoring `parse_config` to return an instance of a `Config` struct

The signature of `parse_config` now indicates that it returns a `Config` value instead of `parse_config`, where we used to return string slices that reference `String`. We define `Config` to contain owned `String` values. The `args` variable in `main` contains the argument values and is only letting the `parse_config` function borrow them. This violates Rust's borrowing rules if `Config` tried to take ownership of the value.

We could manage the `String` data in a number of different ways, but the easiest and most efficient route is to call the `clone` method on the values. This will make a copy of the `Config` instance to own, which takes more time and memory than storing the string data. However, cloning the data also makes our code very straightforward. We have to manage the lifetimes of the references; in this circumstance, giving up some simplicity is a worthwhile trade-off.

The Trade-Offs of Using `clone`

There's a tendency among many Rustaceans to avoid using `clone` to fix `Config::query` because of its runtime cost. In Chapter 13, you'll learn how to use more efficient techniques for this type of situation. But for now, it's okay to copy a few strings to continue using `Config::query` because you'll make these copies only once and your filename and query fields will be copied correctly. It's better to have a working program that's a bit inefficient than to try to fix it on your first pass. As you become more experienced with Rust, it'll be easier to find a more efficient solution, but for now, it's perfectly acceptable to call `clone`.

We've updated `main` so it places the instance of `Config` returned by `parse_config` into a variable named `config`, and we updated the code that previously used the separate `query` and `filename` variables so it now uses the fields on the `Config` struct instead.

Now our code more clearly conveys that `query` and `filename` are related and what they're used for to configure how the program will work. Any code that uses these values knows that they're part of a `Config` instance in the fields named for their purpose.

Creating a Constructor for `Config`

So far, we've extracted the logic responsible for parsing the command line arguments and placed it in the `parse_config` function. Doing so helped us to see that the `Config` and `filename` values were related and that relationship should be conveyed in the `Config` struct. We can use the standard library's `String` type to store the `query` and `filename` values under the same names as struct field names from the `parse_config` function.

So now that the purpose of the `parse_config` function is to create a `Config` instance, we can change `parse_config` from a plain function to a function named `new` that is associated with the `Config` struct. Making this change will make the code more idiomatic. We can use the standard library's `String` type to store the `query` and `filename` values under the same names as struct field names from the `parse_config` function. Listing 12-7 shows the changes we need to make.

Filename: `src/main.rs`

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

    let config = Config::new(&args);

    // --snip--
}

// --snip--

impl Config {
    fn new(args: &[String]) -> Config {
        let query = args[1].clone();
        let filename = args[2].clone();

        Config { query, filename }
    }
}
```

Listing 12-7: Changing `parse_config` into `Config::new`

We've updated `main` where we were calling `parse_config` to instead call `Config::new`. We've also changed the name of `parse_config` to `new` and moved it within an `impl` block that applies to the `Config` type. Try compiling this code again to make sure it works.

Fixing the Error Handling

Now we'll work on fixing our error handling. Recall that attempting to access vector at index 1 or index 2 will cause the program to panic if the vector contains items. Try running the program without any arguments; it will look like this:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep`
thread 'main' panicked at 'index out of bounds: the len is 1
but the index is 1', src/main.rs:29:21
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

The line `index out of bounds: the len is 1 but the index is 1` is an error message for programmers. It won't help our end users understand what happened at all instead. Let's fix that now.

Improving the Error Message

In Listing 12-8, we add a check in the `new` function that will verify that the slice has at least three elements before attempting to access index 1 and 2. If the slice isn't long enough, the program panics with a better error message than the `index out of bounds` message.

Filename: `src/main.rs`

```
// --snip--
fn new(args: &[String]) -> Config {
    if args.len() < 3 {
        panic!("not enough arguments");
    }
// --snip--
```

Listing 12-8: Adding a check for the number of arguments

This code is similar to the `Guess::new` function we wrote in Listing 9-9, where we checked for valid values in the range of 1 to 3. Instead of checking for valid values here, we're checking that the length of `args` is at least 3 and the rest of the code will operate under the assumption that this condition has been met. If `args` has fewer than three elements, this condition will be true, and we call the `panic!` macro to end the program.

With these extra few lines of code in `new`, let's run the program without any arguments to see what the error looks like now:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep`
thread 'main' panicked at 'not enough arguments', src/main.rs:30:1
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

This output is better: we now have a reasonable error message. However, we've included some information we don't want to give to our users. Perhaps using the technique we learned in Chapter 9—returning a `Result` that indicates either success or failure—isn't the best to use here: a call to `panic!` is more appropriate for a program that has encountered a usage problem, as discussed in Chapter 9. Instead, we can use the technique we learned about in Chapter 9—returning a `Result` that indicates either success or failure.

Returning a `Result` from `new` Instead of Calling `panic!`

We can instead return a `Result` value that will contain a `Config` instance if it will describe the problem in the error case. When `config::new` is communicating with `main`, we can use the `Result` type to signal there was a problem. Then we can change `main`'s return type from a variant into a more practical error for our users without the surrounding text and `RUST_BACKTRACE` that a call to `panic!` causes.

Listing 12-9 shows the changes we need to make to the return value of `Config::new`. Note that this won't compile until we've made the changes to `main`, which we'll do in the next listing.

Filename: `src/main.rs`

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

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

        Ok(Config { query, filename })
    }
}
```

Listing 12-9: Returning a `Result` from `Config::new`

Our `new` function now returns a `Result` with a `Config` instance in the success case and a string literal in the error case. Recall from “The Static Lifetime” section in Chapter 4 that `&'static str` is the type of string literals, which is our error message type for this function.

We've made two changes in the body of the `new` function: instead of calling `process::exit` when the user doesn't pass enough arguments, we now return an `Err` value, and we've wrapped the successful return value in an `Ok`. These changes make the function conform to its new return type.

Returning an `Err` value from `Config::new` allows the `main` function to handle the error case by returning from the `new` function and exit the process more cleanly in the error case.

Calling `Config::new` and Handling Errors

To handle the error case and print a user-friendly message, we need to update the code that's being returned by `Config::new`, as shown in Listing 12-10. We'll also update the `main` function to exit the command line tool with a nonzero error code from `panic!` when the user provides an invalid argument. A nonzero exit status is a convention to signal to the process that called our program that it exited with an error state.

Filename: `src/main.rs`

```
use std::process;

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

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

    // --snip--
```

Listing 12-10: Exiting with an error code if creating a new `Config` fails

In this listing, we've used a method we haven't covered before: `unwrap_or_else`. `Result<T, E>` by the standard library. Using `unwrap_or_else` allows us to do non-panic! error handling. If the `Result` is an `Ok` value, this method's behavior is to return the inner value. `Ok` is wrapping. However, if the value is a `Err`, this method calls the code in the *closure*, which is an anonymous function we define as an argument to `unwrap_or_else`. We'll cover closures in more detail in Chapter 12. You need to know that `unwrap_or_else` will pass the inner value of the `Err`, which is a static string "not enough arguments" that we added in Listing 12-9, to our closure `err` that appears between the vertical pipes. The code in the closure can then run when it runs.

We've added a new `use` line to import `process` from the standard library. The code that will be run in the error case is only two lines: we print the `err` value and then call `process::exit`. The `process::exit` function will stop the program immediately with the exit status code. This is similar to the `panic` code used in Listing 12-8, but we no longer get all the extra output. Let's try it:

```
$ cargo run
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.48 secs
Running `target/debug/minigrep`  
Problem parsing arguments: not enough arguments
```

Great! This output is much friendlier for our users.

Extracting Logic from `main`

Now that we've finished refactoring the configuration parsing, let's turn to the code stated in "Separation of Concerns for Binary Projects", we'll extract a function that holds all the logic currently in the `main` function that isn't involved with setting up or handling errors. When we're done, `main` will be concise and easy to verify by being able to write tests for all the other logic.

Listing 12-11 shows the extracted `run` function. For now, we're just making a small improvement of extracting the function. We're still defining the function in `src/main.rs`.

Filename: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    run(config);
}

fn run(config: Config) {
    let contents = fs::read_to_string(config.filename)
        .expect("something went wrong reading the file");

    println!("With text:\n{}", contents);
}

// --snip--
```

Listing 12-11: Extracting a `run` function containing the rest of the program logic

The `run` function now contains all the remaining logic from `main`, starting from the line

`run` function takes the `Config` instance as an argument.

Returning Errors from the `run` Function

With the remaining program logic separated into the `run` function, we can implement error handling, as we did with `Config::new` in Listing 12-9. Instead of allowing the caller to `expect`, the `run` function will return a `Result<T, E>` when something goes wrong. This lets us further consolidate into `main` the logic around handling errors in a useful way. Listing 12-12 shows the changes we need to make to the signature and body of `run`.

Filename: `src/main.rs`

```
use std::error::Error;
// --snip--
fn run(config: Config) -> Result<(), Box
```

Listing 12-12: Changing the `run` function to return `Result`

We've made three significant changes here. First, we changed the return type of `run` to `Result<(), Box<dyn Error>>`. This function previously returned the unit type `()`, but now it returns `Ok()` if everything goes well and `Err()` if there's an error. Note that as the value returned in the `Ok` case, it's the `Box` type, not the `dyn Error` trait object.

For the error type, we used the *trait object* `Box<dyn Error>` (and we've brought it into scope with a `use` statement at the top). We'll cover trait objects in Chapter 20. The `Box` part means that the function will return a type that implements the `Error` trait, but we don't have to specify what particular type the return value will be. This means that the error values that may be of different types in different error cases. The `dyn` part means, it's short for "dynamic."

Second, we've removed the call to `expect` in favor of `?!`, as we talked about in Chapter 12. Instead of `expect`, `?` will return the error value from the current function handle.

Third, the `run` function now returns an `Ok` value in the success case. We've changed the function's success type as `(())` in the signature, which means we need to wrap the `Ok` value. This `Ok(())` syntax might look a bit strange at first, but using it's the idiomatic way to indicate that we're calling `run` for its side effects only; it doesn't return a value.

When you run this code, it will compile but will display a warning:

```
warning: unused `std::result::Result` which must be used
--> src/main.rs:18:5
  |
18 |     run(config);
  |     ^^^^^^^^^^
= note: #[warn(unused_must_use)] on by default
```

Rust tells us that our code ignored the `Result` value and the `Result` value was never checked. But we're not checking to see whether or not there was an error. This reminds us that we probably meant to have some error handling code here!

problem now.

Handling Errors Returned from `run` in `main`

We'll check for errors and handle them using a technique similar to one we used in Listing 12-10, but with a slight difference:

Filename: `src/main.rs`

```
fn main() {
    // --snip--

    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

    if let Err(e) = run(config) {
        println!("Application error: {}", e);

        process::exit(1);
    }
}
```

We use `if let` rather than `unwrap_or_else` to check whether `run` returns `process::exit(1)` if it does. The `run` function doesn't return a value that we can use the same way that `Config::new` returns the `Config` instance. Because `run` returns a unit value in this case, we only care about detecting an error, so we don't need `unwrap_or_else`. We can ignore the unwrapped value because it would only be `()`.

The bodies of the `if let` and the `unwrap_or_else` functions are the same: they both print the error message and call `process::exit` with the error code.

Splitting Code into a Library Crate

Our `minigrep` project is looking good so far! Now we'll split the `src/main.rs` file into the `src/lib.rs` file so we can test it and have a `src/main.rs` file with fewer responsibilities.

Let's move all the code that isn't the `main` function from `src/main.rs` to `src/lib.rs`:

- The `run` function definition
- The relevant `use` statements
- The definition of `Config`
- The `Config::new` function definition

The contents of `src/lib.rs` should have the signatures shown in Listing 12-13 (we've omitted the bodies of the functions for brevity). Note that this won't compile until we move the `main` function to a new file, `src/main.rs`, after this one.

Filename: `src/lib.rs`

```

use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

pub struct Config {
    pub query: String,
    pub filename: String,
}

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

pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    // --snip--
}

```

Listing 12-13: Moving `Config` and `run` into `src/lib.rs`

We've made liberal use of the `pub` keyword: on `Config`, on its fields and its the `run` function. We now have a library crate that has a public API that we can use.

Now we need to bring the code we moved to `src/lib.rs` into the scope of the `src/main.rs`, as shown in Listing 12-14:

Filename: `src/main.rs`

```

extern crate minigrep;

use std::env;
use std::process;

use minigrep::Config;

fn main() {
    // --snip--
    if let Err(e) = minigrep::run(config) {
        // --snip--
    }
}

```

Listing 12-14: Bringing the `minigrep` crate into the scope of `src/main.rs`

To bring the library crate into the binary crate, we use `extern crate minigrep` line to bring the `Config` type into scope, and we prefix it with our crate name. Now all the functionality should be connected and should work correctly. Just run your program with `cargo run` and make sure everything works correctly.

Whew! That was a lot of work, but we've set ourselves up for success in the future. We've made it easier to handle errors, and we've made the code more modular. Almost all of the code is now in `src/lib.rs` from here on out.

Let's take advantage of this newfound modularity by doing something that would be difficult with the old code but is easy with the new code: we'll write some tests!

Developing the Library's Functionality with Test Driven Development

Now that we've extracted the logic into `src/lib.rs` and left the argument collection in `src/main.rs`, it's much easier to write tests for the core functionality of our `search` function directly with various arguments and check return values without having to run the program from the command line. Feel free to write some tests for the functionality in the `search` function on your own.

In this section, we'll add the searching logic to the `minigrep` program by using the test-driven development (TDD) process. This software development technique follows these steps:

1. Write a test that fails and run it to make sure it fails for the reason you expect.
2. Write or modify just enough code to make the new test pass.
3. Refactor the code you just added or changed and make sure the tests continue to pass.
4. Repeat from step 1!

This process is just one of many ways to write software, but TDD can help drive the development process. Writing the test before you write the code that makes the test pass helps to ensure that the code has good coverage throughout the process.

We'll test drive the implementation of the functionality that will actually do the search. We'll start by adding a test that queries the file contents and produces a list of lines that match the query string. We'll implement this functionality in a function called `search`.

Writing a Failing Test

Because we don't need them anymore, let's remove the `println!` statements in `src/main.rs` that we used to check the program's behavior. Then, in `src/lib.rs`, we'll add a test function, as we did in Chapter 11. The test function specifies the `search` function to have: it will take a query and the text to search for the query and return only the lines from the text that contain the query. Listing 12-15 shows this test:

Filename: `src/lib.rs`

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

    #[test]
    fn one_result() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }
}
```

Listing 12-15: Creating a failing test for the `search` function we wish we had

This test searches for the string `"duct"`. The text we're searching is three lines of text that all contain the word `"duct"`. We assert that the value returned from the `search` function is a vector containing the single line `"safe, fast, productive."`, which is what we expect.

We aren't able to run this test and watch it fail because the test doesn't even

function doesn't exist yet! So now we'll add just enough code to get the test to compile: adding a definition of the `search` function that always returns an empty vector. Then the test should compile and fail because an empty vector doesn't contain the line "safe, fast, productive."

Filename: `src/lib.rs`

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    vec![]
}
```

Listing 12-16: Defining just enough of the `search` function so our test will compile.

Notice that we need an explicit lifetime `'a` defined in the signature of `search`. The `contents` argument and the return value. Recall in Chapter 10 that the lifetime `'a` which argument lifetime is connected to the lifetime of the return value. In this example, the returned vector should contain string slices that reference slices of the `contents` argument (rather than the argument `query`).

In other words, we tell Rust that the data returned by the `search` function won't outlive the data passed into the `search` function in the `contents` argument. This is important because a slice needs to be valid for the reference to be valid; if the code makes string slices of `query` rather than `contents`, it will do its safety checks on the wrong data.

If we forget the lifetime annotations and try to compile this function, we'll get an error:

```
error[E0106]: missing lifetime specifier
--> src/lib.rs:5:51
  |
5 | pub fn search(query: &str, contents: &str) -> Vec<&str> {
  |                                     ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but its
  signature does not say whether it is borrowed from `query` or `contents`
```

Rust can't possibly know which of the two arguments we need, so we need to tell it which one is which. In this case, `contents` is the argument that contains all of our text and we want to return a slice of that text. Once we make that match, we know `contents` is the argument that should be connected to the lifetime syntax.

Other programming languages don't require you to connect arguments to references in the function signature. So although this might seem strange, it will get easier over time. You can compare this example with the "Validating References with Lifetimes" section in Chapter 10.

Now let's run the test:

```
$ cargo test
    Compiling minigrep v0.1.0 (file:///projects/minigrep)
--warnings--
    Finished dev [unoptimized + debuginfo] target(s) in 0.43 secs
    Running target/debug/deps/minigrep-abcabcbc

running 1 test
test test::one_result ... FAILED

failures:

---- test::one_result stdout ----
        thread 'test::one_result' panicked at 'assertion failed: `right`'
left: `["safe, fast, productive."]`,
right: `[]')`, src/lib.rs:48:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    test::one_result

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 . 
error: test failed, to rerun pass '--lib'
```

Great, the test fails, exactly as we expected. Let's get the test to pass!

Writing Code to Pass the Test

Currently, our test is failing because we always return an empty vector. To fix this search, our program needs to follow these steps:

- Iterate through each line of the contents.
- Check whether the line contains our query string.
- If it does, add it to the list of values we're returning.
- If it doesn't, do nothing.
- Return the list of results that match.

Let's work through each step, starting with iterating through lines.

Iterating Through Lines with the `lines` Method

Rust has a helpful method to handle line-by-line iteration of strings, conveniently named `lines`. It works as shown in Listing 12-17. Note this won't compile yet:

Filename: `src/lib.rs`

```
pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        // do something with line
    }
}
```

Listing 12-17: Iterating through each line in `contents`

The `lines` method returns an iterator. We'll talk about iterators in depth in Chapter 14, but for now, know that you saw this way of using an iterator in Listing 3-5, where we used a `for` loop to run some code on each item in a collection.

Searching Each Line for the Query

Next, we'll check whether the current line contains our query string. Fortunately, there's a helpful method named `contains` that does this for us! Add a call to the `contains` function, as shown in Listing 12-18. Note this still won't compile yet:

Filename: `src/lib.rs`

```
pub fn search<'a>(query: &'str, contents: &'a str) -> Vec<&'a str> {
    for line in contents.lines() {
        if line.contains(query) {
            // do something with line
        }
    }
}
```

Listing 12-18: Adding functionality to see whether the line contains the string

Storing Matching Lines

We also need a way to store the lines that contain our query string. For that, we can create a vector before the `for` loop and call the `push` method to store a `line` in the vector. After the loop, we return the vector, as shown in Listing 12-19:

Filename: `src/lib.rs`

```
pub fn search<'a>(query: &'str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-19: Storing the lines that match so we can return them

Now the `search` function should return only the lines that contain `query`, a vector of strings. Let's run the test:

```
$ cargo test
--snip--
running 1 test
test test::one_result ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Our test passed, so we know it works!

At this point, we could consider opportunities for refactoring the implementation of the `search` function while keeping the tests passing to maintain the same functionality. The current implementation of the `search` function isn't too bad, but it doesn't take advantage of some useful features from the iterator trait. We'll cover iterators in more detail in Chapter 13, where we'll explore them in detail, and look at how they can make this code even cleaner.

Using the `search` Function in the `run` Function

Now that the `search` function is working and tested, we need to call `search` from the `run` function. We need to pass the `config.query` value and the `contents` that we created earlier:

to the `search` function. Then `run` will print each line returned from `search`

Filename: `src/lib.rs`

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let mut f = File::open(config.filename)?;
    let contents = fs::read_to_string(config.filename)?;

    for line in search(&config.query, &contents) {
        println!("{}", line);
    }

    Ok(())
}
```

We're still using a `for` loop to return each line from `search` and print it.

Now the entire program should work! Let's try it out, first with a word that isn't in the poem, like "frog":

```
$ cargo run frog poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
    Running `target/debug/minigrep frog poem.txt`
How public, like a frog
```

Cool! Now let's try a word that will match multiple lines, like "body":

```
$ cargo run body poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep body poem.txt`
I'm nobody! Who are you?
Are you nobody, too?
How dreary to be somebody!
```

And finally, let's make sure that we don't get any lines when we search for a word that isn't in the poem, such as "monomorphization":

```
$ cargo run monomorphization poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep monomorphization poem.txt`
```

Excellent! We've built our own mini version of a classic tool and learned a lot about file input and output, lifetimes, and closures. We've also learned a bit about line parsing.

To round out this project, we'll briefly demonstrate how to work with environment variables. We'll show how to print to standard error, both of which are useful when you're writing programs.

Working with Environment Variables

We'll improve `minigrep` by adding an extra feature: an option for case-insensitivity. This allows the user to turn on via an environment variable. We could make this feature a command-line argument, but instead we'll use an environment variable. Doing so allows our users to set the environment variable once and have it be case insensitive in that terminal session.

Writing a Failing Test for the Case-Insensitive search Function

We want to add a new `search_case_insensitive` function that we'll call whenever the `case_sensitive` variable is on. We'll continue to follow the TDD process, so the first step is adding a failing test. We'll add a new test for the new `search_case_insensitive` function after the existing test for `one_result` to `case_sensitive` to clarify the differences between the two.

Filename: `src/lib.rs`

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

    #[test]
    fn case_sensitive() {
        let query = "duct";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Duct tape.";

        assert_eq!(
            vec!["safe, fast, productive."],
            search(query, contents)
        );
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

Listing 12-20: Adding a new failing test for the case-insensitive function we're implementing.

Note that we've edited the old test's `contents` too. We've added a new line `"Duct tape."` using a capital D that shouldn't match the query "duct" when using the `case_insensitive` function in a case-sensitive manner. Changing the old test in this way helps ensure that we break the case-sensitive search functionality that we've already implemented now and should continue to pass as we work on the case-insensitive search.

The new test for the `case-insensitive` search uses "rust" as its query. In the `search_case_insensitive` function we're about to add, the query "rUsT" is contained in the `contents` string, containing "Rust:" with a capital R and match the line "Trust me." even though it has different casing than the query. This is our failing test, and it will fail to compile until we've defined the `search_case_insensitive` function. Feel free to add a skeleton implementation of the function; it always returns an empty vector, similar to the way we did for the `search` function in Listing 12-19, so the test will compile and fail.

Implementing the `search_case_insensitive` Function

The `search_case_insensitive` function, shown in Listing 12-21, will be almost identical to the `search_case_sensitive` function. The only difference is that we'll lowercase the `query` and each line in the `contents` before comparing them. Because we lowercase the case of the input arguments, they'll be the same case when we check whether the query matches.

Filename: `src/lib.rs`

```
fn search_case_insensitive<'a>(query: &str, contents: &'a str) -> Vec<String> {
    let query = query.to_lowercase();
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.to_lowercase().contains(&query) {
            results.push(line);
        }
    }

    results
}
```

Listing 12-21: Defining the `search_case_insensitive` function to lowercase both the query and each line before comparing them

First, we lowercase the `query` string and store it in a shadowed variable with `to_lowercase`. Lowercasing the `query` is necessary so no matter whether the user's query is "Rust:", or "rUsT", we'll treat the query as if it were "rust" and be insensitive to the case of the query.

Note that `query` is now a `String` rather than a string slice, because calling `to_lowercase` creates new data rather than referencing existing data. Say the query is "rUsT", as a string slice doesn't contain a lowercase u or t for us to use, so we have to allocate memory for a new string containing "rust". When we pass `query` as an argument to the `contains` method, we add an ampersand because the signature of `contains` is defined to take a `&str`.

Next, we add a call to `to_lowercase` on each `line` before we check whether the line contains the lowercase query. Now that we've converted `line` and `query` to lowercase, the comparison matches no matter what the case of the query is.

Let's see if this implementation passes the tests:

```
running 2 tests
test test::case_insensitive ... ok
test test::case_sensitive ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Great! They passed. Now, let's call the new `search_case_insensitive` function. First, we'll add a configuration option to the `Config` struct to switch between case-sensitive and case-insensitive search. Adding this field will cause the compiler to complain about initializing this field anywhere yet:

Filename: `src/lib.rs`

```
pub struct Config {
    pub query: String,
    pub filename: String,
    pub case_sensitive: bool,
}
```

Note that we added the `case_sensitive` field that holds a Boolean. Next, we want to check the `case_sensitive` field's value and use that to decide whether to call the `search` function or the `search_case_insensitive` function, as shown in Listing 12-2. We can't compile yet:

Filename: `src/lib.rs`

```
pub fn run(config: Config) -> Result<(), Box<dyn Error>> {
    let contents = fs::read_to_string(config.filename)?;

    let results = if config.case_sensitive {
        search(&config.query, &contents)
    } else {
        search_case_insensitive(&config.query, &contents)
    };

    for line in results {
        println!("{}", line);
    }

    Ok(())
}
```

Listing 12-22: Calling either `search` or `search_case_insensitive` based on `config.case_sensitive`

Finally, we need to check for the environment variable. The functions for working with environment variables are in the `env` module in the standard library, so we want to bring it into scope with a `use std::env;` line at the top of `src/lib.rs`. Then we'll use the `var` method on the `env` module to check for an environment variable named `CASE_INSENSITIVE`, as shown in Listing 12-23.

Filename: `src/lib.rs`

```
use std::env;
// --snip--

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

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

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
        Ok(Config { query, filename, case_sensitive })
    }
}
```

Listing 12-23: Checking for an environment variable named `CASE_INSENSITIVE`

Here, we create a new variable `case_sensitive`. To set its value, we call the `env::var` function, passing it the name of the `CASE_INSENSITIVE` environment variable. The `env::var` function returns a `Result` that will be the successful `Ok` variant that contains the value of the environment variable if the environment variable is set. It will return the `Err` variant if the environment variable is not set.

We're using the `is_err` method on the `Result` to check whether it's an error or a success. If it's an error, then `case_sensitive` will be `false`, which means it *should* do a case-sensitive search. If the `CASE_INSENSITIVE` environment variable is set to "true", then `case_sensitive` will be `true`, which means it *should* do a case-insensitive search.

set to anything, `is_err` will return false and the program will perform a case-insensitive search. We don't care about the *value* of the environment variable, just whether it's set or not. Instead of checking `is_err` rather than using `unwrap`, `expect`, or any of the other methods on `Result`.

We pass the value in the `case_sensitive` variable to the `Config` instance so that we can read that value and decide whether to call `search` or `search_case_insensitive`, implemented in Listing 12-22.

Let's give it a try! First, we'll run our program without the environment variable set, which should match any line that contains the word "to" in all lowercase letters:

```
$ cargo run to poem.txt
Compiling minigrep v0.1.0 (file:///projects/minigrep)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
```

Looks like that still works! Now, let's run the program with `CASE_INSENSITIVE=1` and the same query "to".

If you're using PowerShell, you will need to set the environment variable and run the command rather than one:

```
$ $env:CASE_INSENSITIVE=1
$ cargo run to poem.txt
```

We should get lines that contain "to" that might have uppercase letters:

```
$ CASE_INSENSITIVE=1 cargo run to poem.txt
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/minigrep to poem.txt`
Are you nobody, too?
How dreary to be somebody!
To tell your name the livelong day
To an admiring bog!
```

Excellent, we also got lines containing "To"! Our `minigrep` program can now search for lines containing either lowercase or uppercase letters, controlled by an environment variable. Now you know how to make your programs more flexible by accepting either command line arguments or environment variables.

Some programs allow arguments *and* environment variables for the same command. In these cases, the programs decide that one or the other takes precedence. For another example, try controlling case insensitivity through either a command line argument or an environment variable. Decide whether the command line argument or the environment variable has precedence if the program is run with one set to case sensitive and one set to case insensitive.

The `std::env` module contains many more useful features for dealing with environment variables. Check out its documentation to see what is available.

Writing Error Messages to Standard Error Instead of Standard Output

At the moment, we're writing all of our output to the terminal using the `println!` macro. Terminals provide two kinds of output: *standard output* (`stdout`) for general output and *standard error* (`stderr`) for error messages. This distinction enables users to

successful output of a program to a file but still print error messages to the screen.

The `println!` function is only capable of printing to standard output, so we need to print to standard error.

Checking Where Errors Are Written

First, let's observe how the content printed by `minigrep` is currently being written to standard output, including any error messages we want to write to standard error instead of standard output. We can do this by redirecting the standard output stream to a file while also intentionally causing an error. We'll redirect the standard error stream, so any content sent to standard error will end up on the screen.

Command line programs are expected to send error messages to the standard error stream. If we redirect standard output to a file, we can still see error messages on the screen even if we redirect the standard error stream to a file. Our program is not currently well-behaved: we're about to see that it saves its error messages to standard output instead of standard error.

The way to demonstrate this behavior is by running the program with the command `cargo run > output.txt`. This tells the shell to write the contents of standard output to `output.txt` instead of the screen. We didn't see the error message we were expecting printed to the screen, so where did it go? It must have ended up in the file. This is what `output.txt` contains:

```
$ cargo run > output.txt
```

The `>` syntax tells the shell to write the contents of standard output to `output.txt` instead of the screen. We didn't see the error message we were expecting printed to the screen, so where did it go? It must have ended up in the file. This is what `output.txt` contains:

```
Problem parsing arguments: not enough arguments
```

Yup, our error message is being printed to standard output. It's much more common for error messages like this to be printed to standard error so only data from a successful execution ends up in the file. We'll change that.

Printing Errors to Standard Error

We'll use the code in Listing 12-24 to change how error messages are printed. Since we've already refactored `minigrep` earlier in this chapter, all the code that prints error messages is now in the `main` function. The standard library provides the `eprintln!` macro that prints to the standard error stream, so let's change the two places we were calling `println!` to print errors to the standard error stream instead.

Filename: `src/main.rs`

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

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

    if let Err(e) = minigrep::run(config) {
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

Listing 12-24: Writing error messages to standard error instead of standard output

After changing `println!` to `eprintln!`, let's run the program again in the shell with no arguments and redirecting standard output with `>`:

```
$ cargo run > output.txt  
Problem parsing arguments: not enough arguments
```

Now we see the error onscreen and `output.txt` contains nothing, which is the behavior we expect from command line programs.

Let's run the program again with arguments that don't cause an error but still redirect standard output to a file, like so:

```
$ cargo run -t poem.txt > output.txt
```

We won't see any output to the terminal, and `output.txt` will contain our results:

Filename: `output.txt`

```
Are you nobody, too?  
How dreary to be somebody!
```

This demonstrates that we're now using standard output for successful output and standard error for error output as appropriate.

Summary

This chapter recapped some of the major concepts you've learned so far and how to use them to perform common I/O operations in Rust. By using command line arguments and environment variables, and the `eprintln!` macro for printing errors, you're now prepared to handle errors effectively in your applications. By using the concepts in previous chapters, your code will be well-suited to work effectively in the appropriate data structures, handle errors nicely, and be well-documented.

Next, we'll explore some Rust features that were influenced by functional languages: closures and iterators.

Functional Language Features: Iterators and Closures

Rust's design has taken inspiration from many existing languages and techniques, and one of the most significant influences is *functional programming*. Programming in a functional style means treating functions as values by passing them in arguments, returning them from functions, and assigning them to variables for later execution, and so forth.

In this chapter, we won't debate the issue of what functional programming is or isn't. Instead, we'll discuss some features of Rust that are similar to features in many languages that are commonly used in functional programming.

More specifically, we'll cover:

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

Other Rust features, such as pattern matching and enums, which we've covered are influenced by the functional style as well. Mastering closures and iterators will help you write idiomatic, fast Rust code, so we'll devote this entire chapter to them.

Closures: Anonymous Functions that Can Capture Environment

Rust's closures are anonymous functions you can save in a variable or pass around as arguments to other functions. You can create the closure in one place and then call the closure later in a different context. Unlike functions, closures can capture values from the scope they were created in. We'll demonstrate how these closure features allow for code reuse and reduce boilerplate customization.

Creating an Abstraction of Behavior with Closures

Let's work on an example of a situation in which it's useful to store a closure in a variable. Along the way, we'll talk about the syntax of closures, type inference, and trait objects.

Consider this hypothetical situation: we work at a startup that's making an app that helps people exercise. The backend is written in Rust, and the algorithm that generates a workout plan takes into account many factors, such as the app user's age, body type, exercise preferences, recent workouts, and an intensity number they specify. The actual calculation is expensive; what's important is that this calculation takes a few seconds to run. Instead of calling this algorithm every time we need a workout plan, we can store the closure in a variable and only call it once so we don't make the same expensive calculation more often than necessary.

We'll simulate calling this hypothetical algorithm with the function `simulated_expensive_calculation` shown in Listing 13-1, which will print `"calculating slowly..."`, sleep for two seconds, and then return whatever number we passed in:

Filename: `src/main.rs`

```
use std::thread;
use std::time::Duration;

fn simulated_expensive_calculation(intensity: u32) -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    intensity
}
```

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

Next is the `main` function, which contains the parts of the workout app implementation. This function represents the code that the app will call when a user asks for a workout plan. The interaction with the app's frontend isn't relevant to the use of closures, which are just representing inputs to our program and print the outputs.

The required inputs are these:

- An intensity number from the user, which is specified when they run the command line tool. Whether they want a low-intensity workout or a high-intensity workout.
- A random number that will generate some variety in the workout plans.

The output will be the recommended workout plan. Listing 13-2 shows the `main` function.

Filename: src/main.rs

```
fn main() {
    let simulated_user_specified_value = 10;
    let simulated_random_number = 7;

    generate_workout(
        simulated_user_specified_value,
        simulated_random_number
    );
}
```

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

We've hardcoded the variable `simulated_user_specified_value` as 10 and `simulated_random_number` as 7 for simplicity's sake; in an actual program, we'd get the `simulated_user_specified_value` from the app frontend, and we'd use the `rand` crate to generate a random `simulated_random_number`. We did something similar to this in the Guessing Game example in Chapter 2. The `main` function calls a `generate_workout` function with the simulated input values.

Now that we have the context, let's get to the algorithm. The function `generate_workout` in Listing 13-3 contains the business logic of the app that we're most concerned with in this chapter. Most of the code changes in this example will be made to this function.

Filename: src/main.rs

```
fn generate_workout(intensity: u32, random_number: u32) {
    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Next, do {} situps!",
            simulated_expensive_calculation(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}
```

Listing 13-3: The business logic that prints the workout plans based on the intensity level. The `intensity` variable is passed to the `simulated_expensive_calculation` function

The code in Listing 13-3 has multiple calls to the slow calculation function. The `intensity` variable is passed to the `simulated_expensive_calculation` function twice, the `if` inside the outer `else` does not affect the inner one. The code inside the second `else` case calls it once.

The desired behavior of the `generate_workout` function is to first check whether the user wants a low-intensity workout (indicated by a number less than 25) or a high-intensity workout (25 or greater).

Low-intensity workout plans will recommend a number of push-ups and sit-ups. This is a very simple algorithm we're simulating.

If the user wants a high-intensity workout, there's some additional logic: if the number generated by the app happens to be 3, the app will recommend a break, but, the user will get a number of minutes of running based on the complexity.

This code works the way the business wants it to now, but let's say the data is changing and that we need to make some changes to the way we call the `simulated_expensive_calculation` function in the future. To simplify the update when those changes happen, we can move the code so it calls the `simulated_expensive_calculation` function only once. We can do this by extracting the part of the code where we're currently unnecessarily calling the function twice without actually needing both results.

Refactoring Using Functions

We could restructure the workout program in many ways. First, we'll try extracting the calls to the `simulated_expensive_calculation` function into a variable, as shown in Listing 13-4.

Filename: `src/main.rs`

```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_result =
        simulated_expensive_calculation(intensity);

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Next, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result
            );
        }
    }
}
```

Listing 13-4: Extracting the calls to `simulated_expensive_calculation` to one place and storing the result in the `expensive_result` variable

This change unifies all the calls to `simulated_expensive_calculation` and stores the result in a variable. Now, instead of the first `if` block unnecessarily calling the function twice, we're only calling it once and waiting for the result in all cases, which includes the inner `if` block that only handles the case where `random_number` is 3.

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

Refactoring with Closures to Store Code

Instead of always calling the `simulated_expensive_calculation` function before we use its value, we can define a closure and store the *closure* in a variable rather than storing the function itself, as shown in Listing 13-5. We can actually move the whole body of the function into the closure.

simulated_expensive_calculation within the closure we're introducing here.

Filename: src/main.rs

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

Listing 13-5: Defining a closure and storing it in the `expensive_closure` variable

The closure definition comes after the `=` to assign it to the variable `expensive_closure`; we start with a pair of vertical pipes (`|`), inside which we specify the closure; this syntax was chosen because of its similarity to closure definition syntax in JavaScript. This closure has one parameter named `num`: if we had more than one parameter, we would separate them with commas, like `|param1, param2|`.

After the parameters, we place curly brackets that hold the body of the closure. If the closure body is a single expression, the end of the closure, after the closing curly brace, needs a semicolon to complete the `let` statement. The value returned from the last line of code in the closure (`num`) will be the value returned from the closure when it's called, because the closing curly brace ends with a semicolon; just like in function bodies.

Note that this `let` statement means `expensive_closure` contains the *definition* of the function, not the *resulting value* of calling the anonymous function. Recall that closures are “closures” because we want to define the code to call at one point, store that code, and then use that code later. The code we want to call is now stored in `expensive_closure`.

With the closure defined, we can change the code in the `if` blocks to call the closure instead of the function and get the resulting value. We call a closure like we do a function: we use the name that holds the closure definition and follow it with parentheses containing the values we want to use, as shown in Listing 13-6:

Filename: src/main.rs

```

fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}

```

Listing 13-6: Calling the `expensive_closure` we've defined

Now the expensive calculation is called in only one place, and we're only executing the closure once, so we need the results.

However, we've reintroduced one of the problems from Listing 13-3: we're still calling the closure twice in the first `if` block, which will call the expensive code twice and make the program take twice as long as they need to. We could fix this problem by creating a variable local to the function and storing the result of calling the closure, but closures provide us with another solution to this problem. But first let's talk about why there aren't type annotations in closures and the traits involved with closures.

Closure Type Inference and Annotation

Closures don't require you to annotate the types of the parameters or the return type of the closure like regular functions do. Type annotations are required on functions because they're part of the public interface exposed to your users. Defining this interface rigidly is important for everyone to agree on what types of values a function uses and returns. But closures provide an exposed interface like this: they're stored in variables and used without naming them to users of our library.

Closures are usually short and relevant only within a narrow context rather than being part of a public API. Within these limited contexts, the compiler is reliably able to infer the types of the closure's parameters and the return type, similar to how it's able to infer the types of variables.

Making programmers annotate the types in these small, anonymous functions is largely redundant with the information the compiler already has available.

As with variables, we can add type annotations if we want to increase explicitness or decrease the cost of being more verbose than is strictly necessary. Annotating the types for the closure defined in Listing 13-5 would look like the definition shown in Listing 13-7:

Filename: src/main.rs

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

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

With type annotations added, the syntax of closures looks more similar to the function definition in Listing 13-6. The following is a vertical comparison of the syntax for the definition of a function and a closure that has the same behavior. We've added some space between the two columns to make the differences easier to spot. This illustrates how closure syntax is similar to function syntax.

<pre>fn add_one_v1 (x: u32) -> u32 { x + 1 }</pre>	<pre>let add_one_v2 = x: u32 -> u32 { x + 1 };</pre>
<pre>let add_one_v3 = x { x + 1 };</pre>	<pre>let add_one_v4 = x x + 1 ;</pre>

The first line shows a function definition, and the second line shows a fully annotated closure definition. The third line removes the type annotations from the closure definition. The fourth line removes the brackets, which are optional because the closure body has only one expression. These are all valid definitions that will produce the same behavior when they are used.

Closure definitions will have one concrete type inferred for each of their parameters and one inferred type for their return value. For instance, Listing 13-8 shows the definition of a short closure that takes a `String` value it receives as a parameter. This closure isn't very useful except for the sake of example. Note that we haven't added any type annotations to the definition: we can call the closure twice, using a `String` as an argument the first time and a `u32` the second time, and the compiler will catch the error.

Filename: src/main.rs

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

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

The compiler gives us this error:

```
error[E0308]: mismatched types
--> src/main.rs
|
| let n = example_closure(5);
|                               ^ expected struct `std::string::String`
integral variable
|
= note: expected type `std::string::String`
         found type `{integer}`
```

The first time we call `example_closure` with the `String` value, the compiler infers the type of the closure to be `String`. The second time we call `example_closure`, and we get a type error if we try to use a different type with it.

Storing Closures Using Generic Parameters and the `Fn` Traits

Let's return to our workout generation app. In Listing 13-6, our code was still calculating closure more times than it needed to. One option to solve this is to store the expensive closure in a variable for reuse and use the variable in each result, instead of calling the closure again. However, this method could result in code.

Fortunately, another solution is available to us. We can create a struct that wraps the resulting value of calling the closure. The struct will execute the closure each time it is asked for its value, and it will cache the resulting value so the rest of our code doesn't have to be responsible for saving and reusing the result. You may know this pattern as *memoization* or *evaluation*.

To make a struct that holds a closure, we need to specify the type of the closure. The definition needs to know the types of each of its fields. Each closure instance is an anonymous type: that is, even if two closures have the same signature, their memory locations are considered different. To define structs, enums, or function parameters that hold closures, we must use generics and trait bounds, as we discussed in Chapter 10.

The `Fn` traits are provided by the standard library. All closures implement at least one of the `Fn`, `FnMut`, or `FnOnce`. We'll discuss the difference between these traits in the "Environment with Closures" section; in this example, we can use the `Fn` trait.

We add types to the `Fn` trait bound to represent the types of the parameter and return values. The closures we want to store in the struct must have to match this trait bound. In this case, our closure has a `u32` parameter and returns a `u32`, so the trait bound we specify is `Fn(u32) -> u32`.

Listing 13-9 shows the definition of the `Cacher` struct that holds a closure and stores its value:

Filename: src/main.rs

```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

Listing 13-9: Defining a `Cacher` struct that holds a closure in `calculation` and stores its value in `value`

The `Cacher` struct has a `calculation` field of the generic type `T`. The trait bound `Fn` tells the compiler that it's a closure by using the `Fn` trait. Any closure we want to store in the `Cacher` struct must have one `u32` parameter (specified within the parentheses after `Fn`) and must return a `u32` (specified after the `->`).

Note: Functions can implement all three of the `Fn` traits too. If what we want to do requires capturing a value from the environment, we can use a function rather than a closure, where we need something that implements an `Fn` trait.

The `value` field is of type `Option<u32>`. Before we execute the closure, `value` is `None`. When we ask for the value, the `Cacher` asks for the *result* of the closure, the `Cacher` will execute the closure once and store the result within a `Some` variant in the `value` field. Then if the `Cacher` asks for the value again, instead of executing the closure again, the `Cacher` will just return the stored value.

held in the `Some` variant.

The logic around the `value` field we've just described is defined in Listing 13-10.

Filename: `src/main.rs`

```
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Listing 13-10: The caching logic of `Cacher`

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

The `Cacher::new` function takes a generic parameter `T`, which we've defined with a trait bound as the `Cacher` struct. Then `Cacher::new` returns a `Cacher` instance with a closure specified in the `calculation` field and a `None` value in the `value` field because no one has executed the closure yet.

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

If `self.value` is `None`, the code calls the closure stored in `self.calculation`, stores the result in `self.value` for future use, and returns the value as well.

Listing 13-11 shows how we can use this `Cacher` struct in the function `generate_fibonacci`. Listing 13-6:

Filename: `src/main.rs`

```

fn generate_workout(intensity: u32, random_number: u32) {
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            );
        }
    }
}

```

Listing 13-11: Using `Cacher` in the `generate_workout` function to abstract a

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

Try running this program with the `main` function from Listing 13-2. Change the `intensity` and `random_number` variables to different values in the various `if` and `else` blocks, calculating slowly... appears when needed. The `Cacher` takes care of the logic necessary to ensure we aren't recalculating more than we need to so `generate_workout` can focus on what it does best: generating workouts.

Limitations of the `Cacher` Implementation

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

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

```

#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 2);
}

```

This test creates a new `Cacher` instance with a closure that returns the value of `a`. It then calls `c.value(1)` and `c.value(2)`, expecting them to return different values. However, because the closure is captured by the `Cacher` instance, both calls return the same value, `2`.

the `value` method on this `Cacher` instance with an `arg` value of 1 and then we expect the call to `value` with the `arg` value of 2 should return 2.

Run this test with the `Cacher` implementation in Listing 13-9 and Listing 13-10 on the `assert_eq!` with this message:

```
thread 'call_with_different_values' panicked at 'assertion failed:  
  left: `1`,  
  right: `2`', src/main.rs
```

The problem is that the first time we called `c.value` with 1, the `Cacher` instance's `value` closure was stored in `self.value`. Thereafter, no matter what we pass in to the `value` method, it will always return 1.

Try modifying `Cacher` to hold a hash map rather than a single value. The key will be the `arg` values that are passed in, and the values of the hash map will be the closure on that key. Instead of looking at whether `self.value` directly has a value for a given `arg`, the `value` function will look up the `arg` in the hash map and return the value. If the `arg` is not present, the `Cacher` will call the closure and save the resulting value in the hash map along with its `arg` value.

The second problem with the current `Cacher` implementation is that it only takes one parameter of type `u32` and return a `u32`. We might want to cache functions that take a string slice and return `usize` values, for example. To fix this issue, we can add generic parameters to increase the flexibility of the `Cacher` functionality.

Capturing the Environment with Closures

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

Listing 13-12 has an example of a closure stored in the `equal_to_x` variable that refers to a variable from the closure's surrounding environment:

Filename: `src/main.rs`

```
fn main() {  
    let x = 4;  
  
    let equal_to_x = |z| z == x;  
  
    let y = 4;  
  
    assert!(equal_to_x(y));  
}
```

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

Here, even though `x` is not one of the parameters of `equal_to_x`, the closure is still allowed to use the `x` variable that's defined in the same scope that `equal_to_x` is.

We can't do the same with functions; if we try with the following example, or

Filename: `src/main.rs`

```

fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }

    let y = 4;
    assert!(equal_to_x(y));
}

```

We get an error:

```

error[E0434]: can't capture dynamic environment in a fn item; use `fn` closure form instead
} closure form instead
--> src/main.rs
|
4 |     fn equal_to_x(z: i32) -> bool { z == x } ^
|

```

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

When a closure captures a value from its environment, it uses memory to store the closure body. This use of memory is overhead that we don't want to pay where we want to execute code that doesn't capture its environment. Because closures are allowed to capture their environment, defining and using functions will never work.

Closures can capture values from their environment in three ways, which determine how the closure can take a parameter: taking ownership, borrowing mutably, or borrowing immutably. These are encoded in the three `Fn` traits as follows:

- `FnOnce` consumes the variables it captures from its enclosing scope, known as the *environment*. To consume the captured variables, the closure must take ownership of them and move them into the closure when it is defined. The `Once` part of the trait represents the fact that the closure can't take ownership of the same variable more than once, so it can be called only once.
- `FnMut` can change the environment because it mutably borrows values.
- `Fn` borrows values from the environment immutably.

When you create a closure, Rust infers which trait to use based on how the closure moves the captured variables from the environment. All closures implement `FnOnce` because they can all move the captured variables. Closures that don't move the captured variables also implement `FnMut`, and closures that need mutable access to the captured variables also implement `Fn`. In Listing 13-12, the closure borrows `x` immutably (so `equal_to_x` has the `Fn` trait) because the closure only needs to read the value in `x`.

If you want to force the closure to take ownership of the values it uses in the environment, you can use the `move` keyword before the parameter list. This technique is mostly used to move a closure to a new thread to move the data so it's owned by the new thread.

We'll have more examples of `move` closures in Chapter 16 when we talk about moving data between threads. Here's the code from Listing 13-12 with the `move` keyword added to the closure to move the data so it's owned by the new thread.

Filename: `src/main.rs`

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

    let equal_to_x = move |z| z == x;

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

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

    assert!(equal_to_x(y));
}
```

We receive the following error:

```
error[E0382]: use of moved value: `x`
--> src/main.rs:6:40
   |
4 |     let equal_to_x = move |z| z == x;
   |                           ----- value moved (into closure) here
5 |
6 |     println!("can't use x here: {:?}", x);
   |                           ^ value used here after
   |
= note: move occurs because `x` has type `std::vec::Vec<i32>`, which
       doesn't implement the `Copy` trait
```

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

Most of the time when specifying one of the `Fn` trait bounds, you can start with `FnMut` or `FnOnce` based on what happens.

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

Processing a Series of Items with Iterators

The iterator pattern allows you to perform some task on a sequence of items. It's responsible for the logic of iterating over each item and determining when to stop. When you use iterators, you don't have to reimplement that logic yourself.

In Rust, iterators are *lazy*, meaning they have no effect until you call method `next` on them. For example, the code in Listing 13-13 creates an iterator from a vector `v1` by calling the `iter` method defined on `Vec`. This code by itself is not very useful.

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
```

Listing 13-13: Creating an iterator

Once we've created an iterator, we can use it in a variety of ways. In Listing 13-14, we'll see how to use iterators with `for` loops to execute some code on each item, although we haven't yet learned how to `next` did until now.

The example in Listing 13-14 separates the creation of the iterator from the

for loop. The iterator is stored in the `v1_iter` variable, and no iteration tail. When the `for` loop is called using the iterator in `v1_iter`, each element in the vector is printed during one iteration of the loop, which prints out each value.

```
let v1 = vec![1, 2, 3];  
let v1_iter = v1.iter();  
  
for val in v1_iter {  
    println!("Got: {}", val);  
}
```

Listing 13-14: Using an iterator in a `for` loop

In languages that don't have iterators provided by their standard libraries, you can implement the same functionality by starting a variable at index 0, using that variable to index into the vector to get a value, and incrementing the variable value in a loop until it reached the total length of the vector.

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

The Iterator Trait and the `next` Method

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

```
trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // methods with default implementations elided  
}
```

Notice this definition uses some new syntax: `type Item` and `Self::Item`, which are *associated types* with this trait. We'll talk about associated types in depth in Chapter 21, but what you need to know is that this code says implementing the `Iterator` trait requires defining an `Item` type, and this `Item` type is used in the return type of the `next` method. In other words, the `Item` type will be the type returned from the iterator.

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

We can call the `next` method on iterators directly; Listing 13-15 demonstrates how to do that. It also shows that the iterator returned from repeated calls to `next` on the iterator created from the vector is the same iterator.

Filename: `src/lib.rs`

```

#[test]
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}

```

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

Note that we needed to make `v1_iter` mutable: calling the `next` method on an iterator consumes the internal state that the iterator uses to keep track of where it is in the sequence. This code *consumes*, or uses up, the iterator. Each call to `next` eats up an item from the vector. If we didn't need to make `v1_iter` mutable when we used a `for` loop because the iterator takes ownership of `v1` and makes it mutable behind the scenes.

Also note that the values we get from the calls to `next` are immutable references to the items in the vector. The `iter` method produces an iterator over immutable references to the items in the vector. If we want to iterate over mutable references, we can call `iter_mut`. Similarly, if we want to iterate over owned values, we can call `into_iter`.

Methods that Consume the Iterator

The `Iterator` trait has a number of different methods with default implementations in the standard library; you can find out about these methods by looking in the documentation for the `Iterator` trait. Some of these methods call the `next` method on the iterator definition, which is why you're required to implement the `next` method when implementing the `Iterator` trait.

Methods that call `next` are called *consuming adaptors*, because calling them consumes the iterator. One example is the `sum` method, which takes ownership of the iterator and iterates over all the items by repeatedly calling `next`, thus consuming the iterator. As it iterates over the items, it adds each item to a running total and returns the total when iteration is complete. Listing 13-16 shows some code illustrating a use of the `sum` method:

Filename: `src/lib.rs`

```

#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}

```

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

We aren't allowed to use `v1_iter` after the call to `sum` because `sum` takes ownership of the iterator when we call it on.

Methods that Produce Other Iterators

Other methods defined on the `Iterator` trait, known as *iterator adaptors*, allow iterators into different kinds of iterators. You can chain multiple calls to iterator adaptors to perform complex actions in a readable way. But because all iterators are lazy, you have to call consuming adaptor methods to get results from calls to iterator adaptors.

Listing 13-17 shows an example of calling the iterator adaptor method `map`, which takes a closure and calls it on each item to produce a new iterator. The closure here creates a new item from the vector by incrementing it by 1. However, this code produces no output.

Filename: `src/main.rs`

```
let v1: Vec<i32> = vec![1, 2, 3];
v1.iter().map(|x| x + 1);
```

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

The warning we get is this:

```
warning: unused `std::iter::Map` which must be used: iterator adaptors
and do nothing unless consumed
--> src/main.rs:4:5
  |
4 |     v1.iter().map(|x| x + 1);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
= note: #[warn(unused_must_use)] on by default
```

The code in Listing 13-17 doesn't do anything; the closure we've specified never executes. The warning reminds us why: iterator adaptors are lazy, and we need to consume them to get any value out of them.

To fix this and consume the iterator, we'll use the `collect` method, which we saw in Listing 13-16. This method consumes the iterator and collects its elements into a collection data type.

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

Filename: `src/main.rs`

```
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
assert_eq!(v2, vec![2, 3, 4]);
```

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

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

Using Closures that Capture Their Environment

Now that we've introduced iterators, we can demonstrate a common use of their environment by using the `filter` iterator adaptor. The `filter` method closure that takes each item from the iterator and returns a Boolean. If the closure returns `true`, the value will be included in the iterator produced by `filter`. If the closure returns `false`, the value won't be included in the resulting iterator.

In Listing 13-19, we use `filter` with a closure that captures the `shoe_size` environment to iterate over a collection of `Shoe` struct instances. It will return only the specified size.

Filename: src/lib.rs

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

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

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::from("sneaker") },
        Shoe { size: 13, style: String::from("sandal") },
        Shoe { size: 10, style: String::from("boot") },
    ];
    let in_my_size = shoes_in_my_size(shoes, 10);

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

Listing 13-19: Using the `filter` method with a closure that captures `shoe_size`

The `shoes_in_my_size` function takes ownership of a vector of shoes and a `shoe_size` parameter. It returns a vector containing only shoes of the specified size.

In the body of `shoes_in_my_size`, we call `into_iter` to create an iterator that iterates over the vector. Then we call `filter` to adapt that iterator into a new iterator that only yields elements for which the closure returns `true`.

The closure captures the `shoe_size` parameter from the environment and checks each shoe's size, keeping only shoes of the size specified. Finally, calling `collect` on the returned by the adapted iterator into a vector that's returned by the function.

The test shows that when we call `shoes_in_my_size`, we get back only shoes of the size as the value we specified.

Creating Our Own Iterators with the `Iterator` Trait

We've shown that you can create an iterator by calling `iter`, `into_iter`, or `iter_mut`. You can create iterators from the other collection types in the standard library. You can also create iterators that do anything you want by implementing the `Iterator` trait for your own types. As previously mentioned, the only method you're required to provide is the `next` method. Once you've done that, you can use all other methods that are implemented by the `Iterator` trait!

To demonstrate, let's create an iterator that will only ever count from 1 to 5. We'll start by creating a struct to hold some values. Then we'll make this struct into an iterator by implementing the `Iterator` trait and using the values in that implementation.

Listing 13-20 has the definition of the `Counter` struct and an associated `new` function that creates instances of `Counter`:

Filename: `src/lib.rs`

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

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

The `Counter` struct has one field named `count`. This field holds a `u32` value that represents the current value where we are in the process of iterating from 1 to 5. The `count` field is private and its type is `u32`. The implementation of `Counter` provides the implementation of `Iterator` to manage its value. The `new` function enforces that every new instance of `Counter` starts with a value of 0 in the `count` field.

Next, we'll implement the `Iterator` trait for our `Counter` type by defining the `next` method to specify what we want to happen when this iterator is used, as shown in Listing 13-21.

Filename: `src/lib.rs`

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

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

We set the associated `Item` type for our iterator to `u32`, meaning the iterator returns `u32` values. Again, don't worry about associated types yet, we'll cover them in Chapter 18.

We want our iterator to add 1 to the current state, so we initialized `count` to 0 first. If the value of `count` is less than 6, `next` will return the current value via `Some`. If it's 6 or greater, `next` will return `None`.

count is 6 or higher, our iterator will return None.

Using Our Counter Iterator's next Method

Once we've implemented the `Iterator` trait, we have an iterator! Listing 13-22 demonstrates that we can use the iterator functionality of our `Counter` struct method on it directly, just as we did with the iterator created from a vector in Listing 13-1.

Filename: `src/lib.rs`

```
#[test]
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

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

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

Using Other Iterator Trait Methods

We implemented the `Iterator` trait by defining the `next` method, so we can use all the other methods defined in the `Iterator` trait's default implementations as defined in the standard library. These methods all use the `next` method's functionality.

For example, if for some reason we wanted to take the values produced by a `Counter` and pair them with values produced by another `Counter` instance after skipping each pair together, keep only those results that are divisible by 3, and add all of them together, we could do so, as shown in the test in Listing 13-23:

Filename: `src/lib.rs`

```
#[test]
fn using_other_iterator_trait_methods() {
    let sum: u32 = Counter::new().zip(Counter::new().skip(1))
        .map(|(a, b)| a * b)
        .filter(|x| x % 3 == 0)
        .sum();

    assert_eq!(18, sum);
}
```

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

Note that `zip` produces only four pairs; the theoretical fifth pair (`(5, None)`) is omitted because `zip` returns `None` when either of its input iterators return `None`.

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

Improving Our I/O Project

With this new knowledge about iterators, we can improve the I/O project in (iterators to make places in the code clearer and more concise. Let's look at how we can improve our implementation of the `Config::new` function and the `search` function.

Removing a `clone` Using an Iterator

In Listing 12-6, we added code that took a slice of `String` values and created a `Config` struct by indexing into the slice and cloning the values, allowing the code to move ownership of the values from the slice to the `Config` struct. In Listing 13-24, we've reproduced the implementation of the `Config::new` function from Listing 12-23:

Filename: `src/lib.rs`

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

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

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
        if case_sensitive {
            return Err("CASE_INSENSITIVE environment variable must be set");
        }

        Ok(Config { query, filename, case_sensitive })
    }
}
```

Listing 13-24: Reproduction of the `Config::new` function from Listing 12-23

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

We needed `clone` here because we have a slice with `String` elements in the `args` slice, but the `new` function doesn't own `args`. To return ownership of a `Config` instance, we need to move the `String` values from the slice into specific locations in the `Config` struct.

With our new knowledge about iterators, we can change the `new` function to take an iterator as its argument instead of borrowing a slice. We'll use the iterator function to move the `String` values from the slice into the `Config` struct. The code that checks the length of the slice and indexes into specific locations in the `Config` struct is doing this because the iterator will access the value at each index.

Once `Config::new` takes ownership of the iterator and stops using indexing, we can move the `String` values from the iterator into `Config` rather than cloning them, avoiding making a new allocation.

Using the Returned Iterator Directly

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

Filename: `src/main.rs`

```

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

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

    // --snip--
}

```

We'll change the start of the `main` function that we had in Listing 12-24 at to 13-25. This won't compile until we update `Config::new` as well.

Filename: src/main.rs

```

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

    // --snip--
}

```

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

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

Next, we need to update the definition of `Config::new`. In your I/O project's change the signature of `Config::new` to look like Listing 13-26. This still works need to update the function body.

Filename: src/lib.rs

```

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

```

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

The standard library documentation for the `env::args` function shows that it returns is `std::env::Args`. We've updated the signature of the `Config::new` parameter `args` has the type `std::env::Args` instead of `&[String]`. Because ownership of `args` and we'll be mutating `args` by iterating over it, we can add into the specification of the `args` parameter to make it mutable.

Using Iterator Trait Methods Instead of Indexing

Next, we'll fix the body of `Config::new`. The standard library documentation for `std::env::Args` implements the `Iterator` trait, so we know we can call the `next` method. Listing 13-27 updates the code from Listing 12-23 to use the `next` method:

Filename: src/lib.rs

```

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

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

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

        let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

        Ok(Config { query, filename, case_sensitive })
    }
}

```

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

Remember that the first value in the return value of `env::args` is the name we want to ignore that and get to the next value, so first we call `next` and do not care about the value. Second, we call `next` to get the value we want to put in the `query` field. If it returns a `Some`, we use a `match` to extract the value. If it returns `None`, it means no arguments were given and we return early with an `Err` value. We do the same for the `filename` value.

Making Code Clearer with Iterator Adaptors

We can also take advantage of iterators in the `search` function in our I/O program, reproduced here in Listing 13-28 as it was in Listing 12-19:

Filename: src/lib.rs

```

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }

    results
}

```

Listing 13-28: The implementation of the `search` function from Listing 12-19

We can write this code in a more concise way using iterator adaptor methods to avoid having a mutable intermediate `results` vector. The functional programming paradigm minimizes the amount of mutable state to make code clearer. Removing the mutable state enables a future enhancement to make searching happen in parallel, because multiple threads can manage concurrent access to the `results` vector. Listing 13-29 shows this code.

Filename: src/lib.rs

```
pub fn search<'a>(query: &'a str, contents: &'a str) -> Vec<&'a str> {
    contents.lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

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

Recall that the purpose of the `search` function is to return all lines in `contents` that contain the `query`. Similar to the `filter` example in Listing 13-19, this code uses the `filter` iterator adaptor to only keep the lines that `line.contains(query)` returns `true` for. We then collect the resulting lines into another vector with `collect`. Much simpler! Feel free to make the same changes to the `search_case_insensitive` function as well.

The next logical question is which style you should choose in your own code: the loop-based implementation in Listing 13-28 or the version using iterators in Listing 13-29. Many experienced programmers prefer to use the iterator style. It's a bit tougher to get the hang of at first, but once you get a feel for the various iterator adaptors and what they do, iterators can be a great way to express intent. Instead of fiddling with the various bits of looping and building loops by hand, the iterator style focuses on the high-level objective of the loop. This abstracts away some of the details of how the loop works, so it's easier to see the concepts that are unique to this code, such as the filter condition: each element in the iterator must pass the `contains` check.

But are the two implementations truly equivalent? The intuitive assumption is that the iterator-based version will be faster. Let's talk about performance.

Comparing Performance: Loops vs. Iterators

To determine whether to use loops or iterators, you need to know which version is faster: the version with an explicit `for` loop or the version with an iterator.

We ran a benchmark by loading the entire contents of *The Adventures of Sherlock Holmes* by Sir Arthur Conan Doyle into a `String` and looking for the word `the` in the contents. Here are the results of the benchmark on the version of `search` using the `for` loop and the version using an iterator:

```
test bench_search_for ... bench: 19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench: 19,234,900 ns/iter (+/- 657,200)
```

The iterator version was slightly faster! We won't explain the benchmark code in detail, but the point is not to prove that the two versions are equivalent but to get a general idea of how the two implementations compare performance-wise.

For a more comprehensive benchmark, you should check using various text files, different words and words of different lengths as the `query`, and other variations. The point is this: iterators, although a high-level abstraction, get converted into roughly the same code as if you'd written the lower-level code yourself. Iterators are *zero-cost abstractions*, by which we mean using the abstraction imposes no additional overhead. This is analogous to how Bjarne Stroustrup, the original designer of C++, defines *zero-overhead* in "Foundations of C++" (2012):

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

As another example, the following code is taken from an audio decoder. The code uses the linear prediction mathematical operation to estimate future values of an audio signal.

function of the previous samples. This code uses an iterator chain to do some variables in scope: a `buffer` slice of data, an array of 12 `coefficients`, and shift data in `qlp_shift`. We've declared the variables within this example but values; although this code doesn't have much meaning outside of its context, it's a good world example of how Rust translates high-level ideas to low-level code.

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 1..buffer.len() {
    let prediction = coefficients.iter()
        .zip(&buffer[i - 1..i])
        .map(|(&c, &s)| c * s as i64)
        .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

To calculate the value of `prediction`, this code iterates through each of the `coefficients` and uses the `zip` method to pair the coefficient values with `buffer`. Then, for each pair, we multiply the values together, sum all the results, and then shift the sum `qlp_shift` bits to the right.

Calculations in applications like audio decoders often prioritize performance over safety. So, what happens when we run this code? What would this Rust code compile to? Well, as of this writing, it compiles down to assembly that you'd write by hand. There's no loop at all corresponding to the iteration over `coefficients`: Rust knows that there are 12 iterations, so it "unrolls" the loop. This is an optimization that removes the overhead of the loop controlling code and instead generates repetitive code for each iteration of the loop.

All of the coefficients get stored in registers, which means accessing the values is fast and there are no bounds checks on the array access at runtime. All these optimizations that make up this assembly code make the resulting code extremely efficient. Now that you know this, you can use closures without fear! They make code seem like it's higher level but don't incur a performance penalty for doing so.

Summary

Closures and iterators are Rust features inspired by functional programming that contribute to Rust's capability to clearly express high-level ideas at low-level. These implementations of closures and iterators are such that runtime performance is part of Rust's goal to strive to provide zero-cost abstractions.

Now that we've improved the expressiveness of our I/O project, let's look at how to use cargo that will help us share the project with the world.

More About Cargo and Crates.io

So far we've used only the most basic features of Cargo to build, run, and test our project. In this chapter, we'll discuss some of its other, more advanced features. Let's start by looking at what we can do to do the following:

- Customize your build through release profiles
- Publish libraries on [crates.io](#)

- Organize large projects with workspaces
- Install binaries from [crates.io](#)
- Extend Cargo using custom commands

Cargo can do even more than what we cover in this chapter, so for a full exp features, see [its documentation](#).

Customizing Builds with Release Profiles

In Rust, *release profiles* are predefined and customizable profiles with differe allow a programmer to have more control over various options for compiling configured independently of the others.

Cargo has two main profiles: the `dev` profile Cargo uses when you run `cargo build` and the `release` profile Cargo uses when you run `cargo build --release`. The `dev` profile has good defaults for development, and the `release` profile has good defaults for production.

These profile names might be familiar from the output of your builds:

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

The `dev` and `release` shown in this build output indicate that the compiler used different profiles.

Cargo has default settings for each of the profiles that apply when there are no sections in the project's *Cargo.toml* file. By adding `[profile.*]` sections for each profile, you can customize, you can override any subset of the default settings. For example, setting the `opt-level` value for the `dev` and `release` profiles:

Filename: *Cargo.toml*

```
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3
```

The `opt-level` setting controls the number of optimizations Rust will apply. It ranges from 0 to 3. Applying more optimizations extends compilation time, so if you're developing and testing your code often, you'll want faster compilation even if the resulting binary is larger. That is the reason the default `opt-level` for `dev` is 0. When you're ready to release your code, you'll want to spend more time compiling. You'll only compile in `release` mode once, so the `release` mode trades longer compilation time for a smaller, faster program. That is why the default `opt-level` for the `release` profile is 3.

You can override any default setting by adding a different value for it in *Cargo.toml*. If we want to use optimization level 1 in the development profile, we can add the following section to the project's *Cargo.toml* file:

Filename: *Cargo.toml*

```
[profile.dev]
opt-level = 1
```

This code overrides the default setting of 0. Now when we run `cargo build`,

defaults for the `dev` profile plus our customization to `opt-level`. Because Cargo will apply more optimizations than the default, but not as many as in the `release` profile.

For the full list of configuration options and defaults for each profile, see [Cargo's configuration documentation](#).

Publishing a Crate to Crates.io

We've used packages from [crates.io](#) as dependencies of our project, but you can also code with other people by publishing your own packages. The crate registry stores the source code of your packages, so it primarily hosts code that is open source.

Rust and Cargo have features that help make your published package easier to find in the first place. We'll talk about some of these features next and then show how to publish a package.

Making Useful Documentation Comments

Accurately documenting your packages will help other users know how and when it's worth investing the time to write documentation. In Chapter 3, we discussed how to add documentation comments to your code using two slashes, `///`. Rust also has a particular kind of comment for documentation comments that is more convenient as a *documentation comment*, that will generate HTML documentation that displays the contents of documentation comments for public API items instead of just the code. If you're interested in knowing how to *use* your crate as opposed to how your crate is implemented, documentation comments are the way to go.

Documentation comments use three slashes, `///`, instead of two and support all the same features as standard documentation comments, such as support for formatting the text. Place documentation comments just before the item they describe.

Listing 14-1 shows documentation comments for an `add_one` function in a crate.

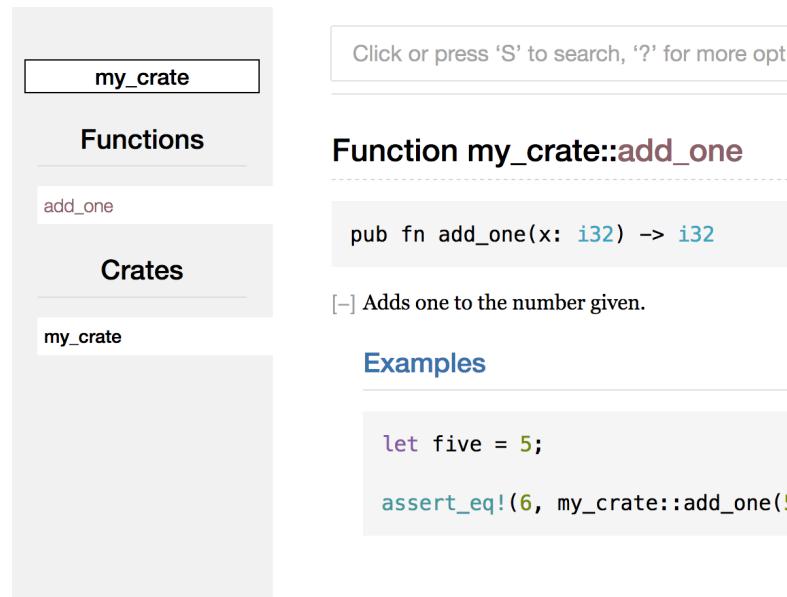
Filename: `src/lib.rs`

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
/// assert_eq!(6, my_crate::add_one(5));
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Listing 14-1: A documentation comment for a function

Here, we give a description of what the `add_one` function does, start a section titled `Examples`, and then provide code that demonstrates how to use the `add_one` function. If you run the `cargo doc` command, Cargo generates the HTML documentation from this documentation comment by running the `rustdoc` tool distributed with Rust and puts the generated documentation in the `target/doc` directory.

For convenience, running `cargo doc --open` will build the HTML for your crate's documentation (as well as the documentation for all of your crate's dependencies) and open the result in a web browser. Navigate to the `add_one` function and you'll see how the documentation comments are rendered, as shown in Figure 14-1:

Figure 14-1: HTML documentation for the `add_one` function

Commonly Used Sections

We used the `# Examples` Markdown heading in Listing 14-1 to create a section title “Examples.” Here are some other sections that crate authors commonly documentation:

- **Panics:** The scenarios in which the function being documented could panic. Function who don’t want their programs to panic should make sure the code handles these situations.
- **Errors:** If the function returns a `Result`, describing the kinds of errors and what conditions might cause those errors to be returned can be helpful. Write code to handle the different kinds of errors in different ways.
- **Safety:** If the function is `unsafe` to call (we discuss unsafety in Chapter 14), a section explaining why the function is unsafe and covering the invariant it expects callers to uphold.

Most documentation comments don’t need all of these sections, but this is a good way to remind you of the aspects of your code that people calling your code will be interested in.

Documentation Comments as Tests

Adding example code blocks in your documentation comments can help demonstrate how your library works. Doing so has an additional bonus: running `cargo test` will run the examples in your documentation as tests! Nothing is better than documentation with examples that work than examples that don’t work because the code has changed since they were written. If we run `cargo test` with the documentation for the `add_one` function from Listing 14-1, we will see a section in the test results like this:

```

Doc-tests my_crate

running 1 test
test src/lib.rs - add_one (line 5) ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

Now if we change either the function or the example so the `assert_eq!` in the test fails, run `cargo test` again, we'll see that the doc tests catch that the example and the function don't sync with each other!

Commenting Contained Items

Another style of doc comment, `///!`, adds documentation to the item that contains the comment, rather than adding documentation to the items following the comments. We can use `///!` comments inside the crate root file (`src/lib.rs` by convention) or inside a module or a crate as a whole.

For example, if we want to add documentation that describes the purpose of the crate, or the module, that contains the `add_one` function, we can add documentation comments to the beginning of the `src/lib.rs` file, as shown in Listing 14-2:

Filename: `src/lib.rs`

```

///! # My Crate
///!
///! `my_crate` is a collection of utilities to make performing certain
///! calculations more convenient.

/// Adds one to the number given.
// --snip--

```

Listing 14-2: Documentation for the `my_crate` crate as a whole

Notice there isn't any code after the last line that begins with `///!`. Because the documentation for the entire crate is contained in the `src/lib.rs` file, we're documenting the item that contains the entire crate, not just the function.

When we run `cargo doc --open`, these comments will display on the front page of the documentation for `my_crate` above the list of public items in the crate, as shown in Figure 14-2.

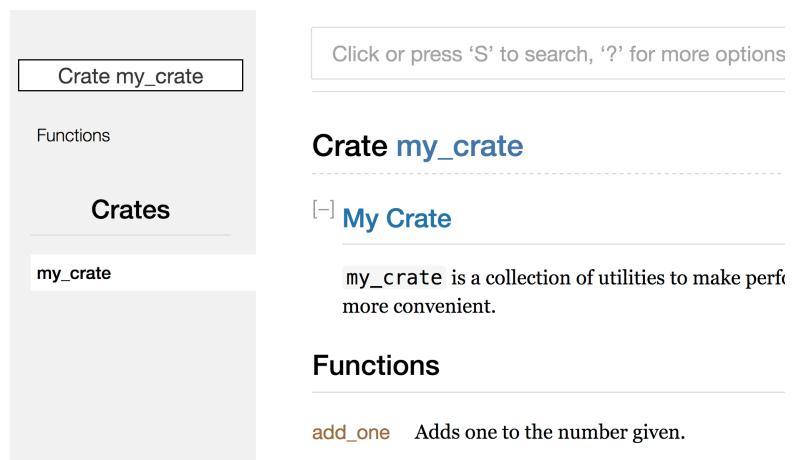


Figure 14-2: Rendered documentation for `my_crate`, including the comments from Listing 14-2.

a whole

Documentation comments within items are useful for describing crates and them to explain the overall purpose of the container to help your users understand organization.

Exporting a Convenient Public API with `pub use`

In Chapter 7, we covered how to organize our code into modules using the `mod` keyword, how to make items public using the `pub` keyword, and how to bring items into a scope using the `use` keyword. However, the structure that makes sense to you while you're developing your code might not be very convenient for your users. You might want to organize your structs into multiple levels, but then people who want to use a type you've defined deep in your crate will have trouble finding out that type exists. They might also be annoyed at having to write `my_crate::some_module::another_module::UsefulType`; rather than `use my_crate::some_module::another_module::UsefulType`.

The structure of your public API is a major consideration when publishing a crate. If your crate has a large module hierarchy, users who are less familiar with the structure than you are and might have trouble finding the pieces they want to use if your crate has a large module hierarchy.

The good news is that if the structure *isn't* convenient for others to use from your crate, you don't have to rearrange your internal organization: instead, you can re-export an item from a module that's different from your private structure by using `pub use`. Re-exporting an item in one location and makes it public in another location, as if it were defined in that location instead.

For example, say we made a library named `art` for modeling artistic concepts. Inside the crate are two modules: a `kinds` module containing two enums named `PrimaryColor` and `SecondaryColor`, and a `utils` module containing a function named `mix`, as shown below:

Filename: `src/lib.rs`

```
///! # Art
///!
///! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
    }
}
```

Listing 14-3: An `art` library with items organized into `kinds` and `utils` modules.

Figure 14-3 shows what the front page of the documentation for this crate might look like:

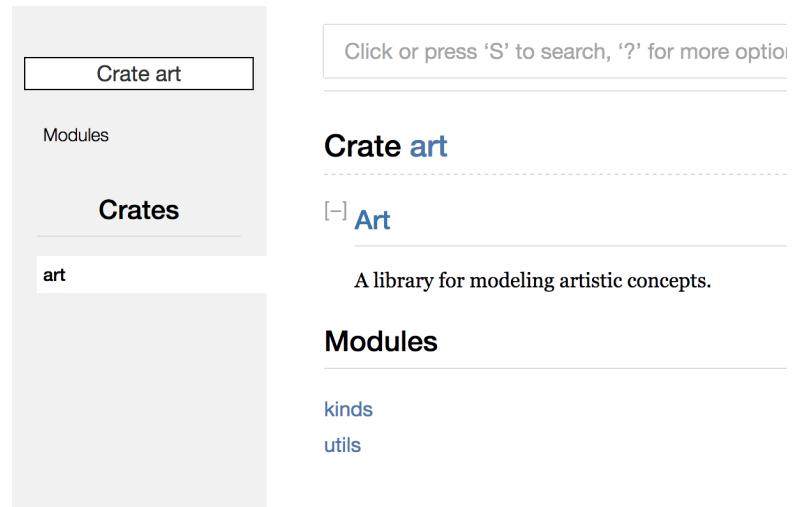


Figure 14-3: Front page of the documentation for `art` that lists the `kinds` and `utils` modules.

Note that the `PrimaryColor` and `SecondaryColor` types aren't listed on the `mix` function. We have to click `kinds` and `utils` to see them.

Another crate that depends on this library would need `use` statements that import items from the `art` crate, specifying the module structure that's currently defined. Listing 14-4 shows how to use the `PrimaryColor` and `mix` items from the `art` crate:

Filename: `src/main.rs`

```
extern crate art;

use art::kinds::PrimaryColor;
use art::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red, yellow);
}
```

Listing 14-4: A crate using the `art` crate's items with its internal structure exposed.

The author of the code in Listing 14-4, which uses the `art` crate, had to figure out where to look for the `PrimaryColor` type. It's in the `kinds` module and `mix` is in the `utils` module. This makes it less convenient to use the crate because the `art` crate is more relevant to developers working on the `art` crate than to users of the `art` crate. The internal structure that organizes parts of the crate into the `kinds` and `utils` module doesn't contain any useful information for someone trying to use the `art` crate. Instead, the `art` crate's module structure causes confusion because it's not clear where to look for the `PrimaryColor` type given the module names in the `use` statements.

To remove the internal organization from the public API, we can modify the `src/lib.rs` file in Listing 14-3 to add `pub` `use` statements to re-export the items at the top level, as shown in Listing 14-5.

Filename: `src/lib.rs`

```
///! # Art
///
///! A library for modeling artistic concepts.

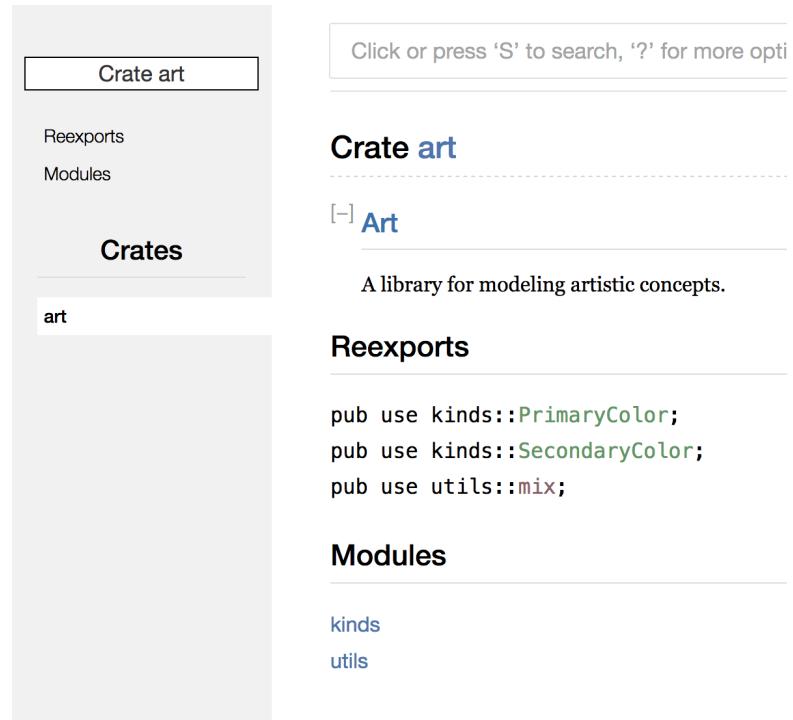
pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;

pub mod kinds {
    // --snip--
}

pub mod utils {
    // --snip--
}
```

Listing 14-5: Adding `pub use` statements to re-export items

The API documentation that `cargo doc` generates for this crate will now list the front page, as shown in Figure 14-4, making the `PrimaryColor` and `SecondaryColor` and the `mix` function easier to find.

Figure 14-4: The front page of the documentation for `art` that lists the re-exported items.

The `art` crate users can still see and use the internal structure from Listing 14-4, or they can use the more convenient structure in Listing 14-5, as shown in Figure 14-4.

Filename: src/main.rs

```
extern crate art;

use art::PrimaryColor;
use art::mix;

fn main() {
    // --snip--
}
```

Listing 14-6: A program using the re-exported items from the `art` crate

In cases where there are many nested modules, re-exporting the types at the top level of a module can make a significant difference in the experience of people who use the crate.

Creating a useful public API structure is more of an art than a science, and you will need to experiment to find an API that works best for your users. Choosing `pub use` gives you flexibility in how you structure your crate internally and decouples that internal structure from what you present to the world. You can even show some of the code of crates you've installed to see if their internal structure does not match what you expect of their public API.

Setting Up a [Crates.io](#) Account

Before you can publish any crates, you need to create an account on [crates.io](#). To do so, visit the home page at [crates.io](#) and log in via a GitHub account. (This is currently a requirement, but the site might support other ways of creating accounts in the future.) Once you're logged in, visit your account settings at <https://crates.io/me/> and generate a new token.

Then run the `cargo login` command with your API key, like this:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

This command will inform Cargo of your API token and store it locally in `~/.cargo/config`. Note that this token is a *secret*: do not share it with anyone else. If you do share it, revoke it immediately. If for some reason, you should revoke it and generate a new token on [crates.io](#).

Adding Metadata to a New Crate

Now that you have an account, let's say you have a crate you want to publish. To do that, you'll need to add some metadata to your crate by adding it to the `[package]` section of your `Cargo.toml` file.

Your crate will need a unique name. While you're working on a crate locally, you can give it whatever name you'd like. However, crate names on [crates.io](#) are allocated on a first-come, first-served basis. Once a crate name is taken, no one else can publish a crate with that name. If you want to use a name that you don't own, you can add the `[package]` section to your `Cargo.toml` file under `[package]` to use the name for publishing, like so:

Filename: `Cargo.toml`

```
[package]
name = "guessing_game"
```

Even if you've chosen a unique name, when you run `cargo publish` to publish your crate to the public point, you'll get a warning and then an error:

```
$ cargo publish
    Updating registry `https://github.com/rust-lang/crates.io-index'
warning: manifest has no description, license, license-file, documentation,
homepage or repository.
--snip--
error: api errors: missing or empty metadata fields: description, license
```

The reason is that you're missing some crucial information: a description and a license. People will know what your crate does and under what terms they can use it if you need to include this information in the `Cargo.toml` file.

Add a description that is just a sentence or two, because it will appear with your results. For the `license` field, you need to give a *license identifier value*. The [Software Package Data Exchange \(SPDX\)](#) lists the identifiers you can use for this. To specify that you've licensed your crate using the MIT License, add the `MIT` identifier:

Filename: `Cargo.toml`

```
[package]
name = "guessing_game"
license = "MIT"
```

If you want to use a license that doesn't appear in the SPDX, you need to place the license in a file, include the file in your project, and then use `license-file` to point to that file instead of using the `license` key.

Guidance on which license is appropriate for your project is beyond the scope of this book, but it's important to note that people in the Rust community license their projects in the same way as Rust itself: `MIT OR Apache-2.0`. This practice demonstrates that you can also specify multiple licenses for your project by separating them with `OR`.

With a unique name, the version, the author details that `cargo new` added to your crate, your description, and a license added, the `Cargo.toml` file for a project might look like this:

Filename: `Cargo.toml`

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
description = "A fun game where you guess what number the computer thinks of"
license = "MIT OR Apache-2.0"

[dependencies]
```

[Cargo's documentation](#) describes other metadata you can specify to ensure your crate is more easily used.

Publishing to Crates.io

Now that you've created an account, saved your API token, chosen a name for your crate, and specified the required metadata, you're ready to publish! Publishing a crate makes it available to others to use.

Be careful when publishing a crate because a publish is *permanent*. The version cannot be overwritten, and the code cannot be deleted. One major goal of [crates.io](#) is to build an archive of code so that builds of all projects that depend on crates from [crates.io](#) work. Allowing version deletions would make fulfilling that goal impossible. If you publish a crate, you should keep it up-to-date with bug fixes and improvements.

limit to the number of crate versions you can publish.

Run the `cargo publish` command again. It should succeed now:

```
$ cargo publish
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Packaging guessing_game v0.1.0 (file:///projects/guessing_game)
  Verifying guessing_game v0.1.0 (file:///projects/guessing_game)
  Compiling guessing_game v0.1.0
    (file:///projects/guessing_game/target/package/guessing_game-0.1.0)
      Finished dev [unoptimized + debuginfo] target(s) in 0.19 secs
  Uploading guessing_game v0.1.0 (file:///projects/guessing_game)
```

Congratulations! You've now shared your code with the Rust community, and your crate as a dependency of their project.

Publishing a New Version of an Existing Crate

When you've made changes to your crate and are ready to release a new version value specified in your `Cargo.toml` file and republish. Use the `Sema` decide what an appropriate next version number is based on the kinds of changes. Then run `cargo publish` to upload the new version.

Removing Versions from Crates.io with `cargo yank`

Although you can't remove previous versions of a crate, you can prevent any adding them as a new dependency. This is useful when a crate version is broken by another. In such situations, Cargo supports *yanking* a crate version.

Yanking a version prevents new projects from starting to depend on that version existing projects that depend on it to continue to download and depend on it. A yank means that all projects with a `Cargo.lock` will not break, and any future generated will not use the yanked version.

To yank a version of a crate, run `cargo yank` and specify which version you

```
$ cargo yank --vers 1.0.1
```

By adding `--undo` to the command, you can also undo a yank and allow projects on a version again:

```
$ cargo yank --vers 1.0.1 --undo
```

A yank *does not* delete any code. For example, the yank feature is not intended to accidentally uploaded secrets. If that happens, you must reset those secrets.

Cargo Workspaces

In Chapter 12, we built a package that included a binary crate and a library crate. If you develops, you might find that the library crate continues to get bigger and you want to package further into multiple library crates. In this situation, Cargo offers a feature that can help manage multiple related packages that are developed in tandem.

Creating a Workspace

A *workspace* is a set of packages that share the same `Cargo.lock` and output code for a project using a workspace—we'll use trivial code so we can concentrate on the workspace. There are multiple ways to structure a workspace; we're going to do it the way. We'll have a workspace containing a binary and two libraries. The binary will have most of the main functionality, will depend on the two libraries. One library will provide an `add_one` function, and a second library an `add_two` function. These three crates will be part of the workspace.

```
$ mkdir add
$ cd add
```

Next, in the `add` directory, we create the `Cargo.toml` file that will configure the workspace. This file won't have a `[package]` section or the metadata we've seen in other `Cargo.toml` files. Instead, it will start with a `[workspace]` section that will allow us to add members to the workspace by specifying the path to our binary crate; in this case, that path is `adder`:

Filename: `Cargo.toml`

```
[workspace]

members = [
    "adder",
]
```

Next, we'll create the `adder` binary crate by running `cargo new` within the `adder` directory:

```
$ cargo new adder
   Created binary (application) `adder` project
```

At this point, we can build the workspace by running `cargo build`. The files in the workspace should look like this:

```
├── Cargo.lock
├── Cargo.toml
└── adder
    ├── Cargo.toml
    └── src
        └── main.rs
└── target
```

The workspace has one `target` directory at the top level for the compiled artifacts. If we were to run `cargo build` inside the `adder` directory, the compiled artifacts would still end up in `adder/target`, which is not what we want. In a workspace, crates are meant to depend on each other. If each crate had its own `target` directory, then every time we ran `cargo build` on the workspace, each crate would have to recompile each of the other crates in the workspace to link them together. By sharing one `target` directory, the crates can avoid unnecessary rebuilds.

Creating the Second Crate in the Workspace

Next, let's create another member crate in the workspace and call it `add-one`. We'll update the `Cargo.toml` to specify the `add-one` path in the `members` list:

Filename: `Cargo.toml`

[workspace]

```
members = [
    "adder",
    "add-one",
]
```

Then generate a new library crate named `add-one`:

```
$ cargo new add-one --lib
Created library `add-one` project
```

Your `add` directory should now have these directories and files:

```
└── Cargo.lock
└── Cargo.toml
└── add-one
    ├── Cargo.toml
    └── src
        └── lib.rs
└── adder
    ├── Cargo.toml
    └── src
        └── main.rs
└── target
```

In the `add-one/src/lib.rs` file, let's add an `add_one` function:

Filename: `add-one/src/lib.rs`

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Now that we have a library crate in the workspace, we can have the binary depend on the library crate `add-one`. First, we'll need to add a path dependency on `add-one/Cargo.toml`.

Filename: `adder/Cargo.toml`

[dependencies]

```
add-one = { path = "../add-one" }
```

Cargo doesn't assume that crates in a workspace will depend on each other, so we need to be explicit about the dependency relationships between the crates.

Next, let's use the `add_one` function from the `add-one` crate in the `adder` crate by adding it to `/main.rs` file and add an `extern crate` line at the top to bring the new `add-one` crate into scope. Then change the `main` function to call the `add_one` function, as in Listing 14-7.

Filename: `adder/src/main.rs`

```
extern crate add-one;

fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}
```

Listing 14-7: Using the `add-one` library crate from the `adder` crate

Let's build the workspace by running `cargo build` in the top-level `add` directory:

```
$ cargo build
    Compiling add-one v0.1.0 (file:///projects/add/add-one)
    Compiling adder v0.1.0 (file:///projects/add/adder)
        Finished dev [unoptimized + debuginfo] target(s) in 0.68 secs
```

To run the binary crate from the `add` directory, we need to specify which package we want to use by using the `-p` argument and the package name with `cargo run`:

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
        Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

This runs the code in `adder/src/main.rs`, which depends on the `add-one` crate.

Depending on an External Crate in a Workspace

Notice that the workspace has only one `Cargo.lock` file at the top level of the workspace, and not each individual crate having a `Cargo.lock` in each crate's directory. This ensures that all crates are compatible with each other and have the same version of all dependencies. If we add the `rand` crate to the `adder/Cargo.toml` and `add-one/Cargo.toml` files, Cargo will resolve both of those to one version of `rand` and record that in the workspace's `Cargo.lock` file. Making all crates in the workspace use the same dependencies means the code in each crate will always be compatible with each other. Let's add the `rand` crate to the `add-one` crate by adding it to the `[dependencies]` section in the `add-one/Cargo.toml` file to be able to use the `rand` crate in the `add-on` crate.

Filename: `add-one/Cargo.toml`

```
[dependencies]
rand = "0.3.14"
```

We can now add `extern crate rand;` to the `add-one/src/lib.rs` file, and build the workspace by running `cargo build` in the `add` directory. This will bring in and compile the `rand` crate for us.

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Downloading rand v0.3.14
    --snip--
    Compiling rand v0.3.14
    Compiling add-one v0.1.0 (file:///projects/add/add-one)
    Compiling adder v0.1.0 (file:///projects/add/adder)
    Finished dev [unoptimized + debuginfo] target(s) in 10.18 secs
```

The top-level `Cargo.lock` now contains information about the dependency of the `rand` crate. However, even though `rand` is used somewhere in the workspace, we can't build the workspace unless we add `rand` to their `Cargo.toml` files as well. For example, if we add `rand` to the `adder/Cargo.toml` file and then add `extern crate rand;` to the `adder/src/main.rs` file for the `adder` crate, we'll get an error:

```
$ cargo build
    Compiling adder v0.1.0 (file:///projects/add/adder)
error: use of unstable library feature 'rand': use `rand` from crate `rand` (see issue #27703)
--> adder/src/main.rs:1:1
| 
1 | extern crate rand;
```

To fix this, edit the `Cargo.toml` file for the `adder` crate and indicate that `rand` is a dependency for that crate as well. Building the `adder` crate will add `rand` to the list of dependencies.

Cargo.lock, but no additional copies of `rand` will be downloaded. Cargo has one copy of `rand` in the workspace using the `rand` crate will be using the same version. Using one copy of `rand` across the workspace saves space because we won't have multiple copies of the crate. All the crates in the workspace will be compatible with each other.

Adding a Test to a Workspace

For another enhancement, let's add a test of the `add_one::add_one` function to the `add-one` crate:

Filename: `add-one/src/lib.rs`

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}

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

    #[test]
    fn it_works() {
        assert_eq!(3, add_one(2));
    }
}
```

Now run `cargo test` in the top-level `add` directory:

```
$ cargo test
Compiling add-one v0.1.0 (file:///projects/add/add-one)
Compiling adder v0.1.0 (file:///projects/add/adder)
Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
    Running target/debug/deps/add_one-f0253159197f7841

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

    Running target/debug/deps/adder-f88af9d2cc175a5e

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
```

The first section of the output shows that the `it_works` test in the `add-one` section shows that zero tests were found in the `adder` crate, and then the last section shows that zero tests were found in the `add-one` crate. Running `cargo test` in a workspace like this one will run the tests for all the crates in the workspace.

We can also run tests for one particular crate in a workspace from the top-level directory by running `cargo test` with the `-p` flag and specifying the name of the crate we want to test:

```
$ cargo test -p add-one
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running target/debug/deps/add_one-b3235fea9a156f74

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Doc-tests add-one

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

This output shows `cargo test` only ran the tests for the `add-one` crate and its crate tests.

If you publish the crates in the workspace to <https://crates.io/>, each crate in the workspace will be published separately. The `cargo publish` command does not have an `--all` flag, so you must change to each crate's directory and run `cargo publish` or publish the entire workspace to publish the crates.

For additional practice, add an `add-two` crate to this workspace in a similar way to the `add-one` crate!

As your project grows, consider using a workspace: it's easier to understand the components than one big blob of code. Furthermore, keeping the crates in a workspace makes coordination between them easier if they are often changed at the same time.

Installing Binaries from Crates.io with `cargo install`

The `cargo install` command allows you to install and use binary crates located on crates.io. It's meant to be a convenient way for Rust developers to replace system packages; it's meant to be a convenient way for Rust developers to use binary crates that others have shared on crates.io. Note that you can only install packages that have a *binary target*. A *binary target* is the runnable program that is created if the crate has another file specified as a binary, as opposed to a library target that isn't run by `cargo build`. Binary targets are usually suitable for including within other programs. Usually, crates have information about whether a crate is a library, has a binary target, or both.

All binaries installed with `cargo install` are stored in the installation root's `target` directory. If you installed Rust using `rustup.rs` and don't have any custom configurations, this directory is `~/.cargo/bin`. Ensure that directory is in your `$PATH` to be able to run programs installed with `cargo install`.

For example, in Chapter 12 we mentioned that there's a Rust implementation of `grep` called `ripgrep` for searching files. If we want to install `ripgrep`, we can run the following command:

```
$ cargo install ripgrep
Updating registry `https://github.com/rust-lang/crates.io-index'
Downloaded ripgrep v0.3.2
--snip--
Compiling ripgrep v0.3.2
    Finished release [optimized + debuginfo] target(s) in 97.91 seconds
Installing ~/cargo/bin/rg
```

The last line of the output shows the location and the name of the installed binary: the name of the `ripgrep` crate is `rg`. As long as the installation directory is in your `$PATH`, as noted above, you can run it like any other command.

you can then run `rg --help` and start using a faster, rustier tool for searchi

Extending Cargo with Custom Commands

Cargo is designed so you can extend it with new subcommands without having binary in your `$PATH` is named `cargo-something`, you can run it as if it was by running `cargo something`. Custom commands like this are also listed when running `cargo --list`. Being able to use `cargo install` to install extensions and tools built on top of the built-in Cargo tools is a super convenient benefit of Cargo's design!

Summary

Sharing code with Cargo and [crates.io](#) is part of what makes the Rust ecosystem great for building large systems. Rust's standard library is small and stable, but crates are easy to use and can be updated independently. You can improve on a timeline different from that of the language. Don't be shy about sharing your code! If you find a crate useful to you on [crates.io](#); it's likely that it will be useful to someone else as well.

Smart Pointers

A *pointer* is a general concept for a variable that contains an address in memory, or "points at," some other data. The most common kind of pointer in Rust is the `&` pointer, which you learned about in Chapter 4. References are indicated by the `&` symbol and are pointers to the data they point to. They don't have any special capabilities other than referring to the data they point to and are the kind of pointer we use most often.

Smart pointers, on the other hand, are data structures that not only act like a pointer but also provide additional metadata and capabilities. The concept of smart pointers isn't unique to Rust; it originated in C++ and exist in other languages as well. In Rust, the `Box`, `Vec`, and `String` types defined in the standard library provide functionality beyond that provided by the `&` pointer. A good example that we'll explore in this chapter is the *reference counting* smart pointer, which enables you to have multiple owners of data by keeping track of the number of references to the data and automatically cleaning up the data when no owners remain, freeing up memory.

In Rust, which uses the concept of ownership and borrowing, an additional characteristic of smart pointers compared to references and raw pointers is that references are pointers that only borrow the data they point to, while smart pointers own the data they point to.

We've already encountered a few smart pointers in this book, such as `String` and `Vec`. We first saw them in Chapter 8, although we didn't call them smart pointers at the time. Both the `String` and `Vec` types are smart pointers because they own some memory and allow you to manipulate it. They implement the `Clone` trait (such as their capacity) and extra capabilities or guarantees (such as with `String`) that ensure the data will always be valid UTF-8.

Smart pointers are usually implemented using structs. The characteristic that distinguishes a smart pointer from an ordinary struct is that smart pointers implement the `Deref` trait. The `Deref` trait allows an instance of the smart pointer struct to behave like a reference to the data it owns, allowing code that works with either references or smart pointers. The `Drop` trait allows smart pointers to run cleanup code that is run when an instance of the smart pointer goes out of scope. In this chapter, we'll discuss both traits and demonstrate why they're important to smart pointer safety.

Given that the smart pointer pattern is a general design pattern used frequently in Rust, this chapter won't cover every existing smart pointer. Many libraries have their own smart pointer implementations.

even write your own. We'll cover the most common smart pointers in the standard library:

- `Box<T>` for allocating values on the heap
- `Rc<T>`, a reference counting type that enables multiple ownership
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces borrowing rules at runtime instead of compile time

In addition, we'll cover the *interior mutability* pattern where an immutable type can be used for mutating an interior value. We'll also discuss *reference cycles*: how they can lead to memory leaks and how to prevent them.

Let's dive in!

Using `Box<T>` to Point to Data on the Heap

The most straightforward smart pointer is a *box*, whose type is written `Box<T>`. A `Box` stores data on the heap rather than the stack. What remains on the stack is a pointer to the data. Refer to Chapter 4 to review the difference between the stack and the heap.

Boxes don't have performance overhead, other than storing their data on the heap instead of the stack. But they don't have many extra capabilities either. You'll use them most often in these situations:

- When you have a type whose size can't be known at compile time and you need to store that type in a context that requires an exact size
- When you have a large amount of data and you want to transfer ownership of it. The data won't be copied when you do so
- When you want to own a value and you care only that it's a type that implements the `Clone` trait rather than being of a specific type

We'll demonstrate the first situation in the "Enabling Recursive Types with Boxes." In the second case, transferring ownership of a large amount of data can take a long time if the data is copied around on the stack. To improve performance in this situation, we can store a small amount of data on the stack and a large amount of data on the heap in a box. Then, only the small amount of pointer data remains on the stack, while the data it references stays in one place on the heap. The *trait object*, and Chapter 17 devotes an entire section, "Using Trait Objects That Work with Different Types," just to that topic. So what you learn here you'll apply again.

Using a `Box<T>` to Store Data on the Heap

Before we discuss this use case for `Box<T>`, we'll cover the syntax and how to use it to store data within a `Box<T>`.

Listing 15-1 shows how to use a box to store an `i32` value on the heap:

Filename: `src/main.rs`

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Listing 15-1: Storing an `i32` value on the heap using a box

We define the variable `b` to have the value of a `Box` that points to the value `5` on the heap. This program will print `b = 5`; in this case, we can access the data stored in the heap directly.

how we would if this data were on the stack. Just like any owned value, when its scope, as `b` does at the end of `main`, it will be deallocated. The deallocation (stored on the stack) and the data it points to (stored on the heap).

Putting a single value on the heap isn't very useful, so you won't use boxes `b` very often. Having values like a single `i32` on the stack, where they're stored, is appropriate in the majority of situations. Let's look at a case where boxes all that we wouldn't be allowed to if we didn't have boxes.

Enabling Recursive Types with Boxes

At compile time, Rust needs to know how much space a type takes up. One type known at compile time is a *recursive type*, where a value can have as part of it the same type. Because this nesting of values could theoretically continue indefinitely, Rust needs to know how much space a value of a recursive type needs. However, boxes help with this: by inserting a box in a recursive type definition, you can have recursive types.

Let's explore the *cons list*, which is a data type common in functional programming. An example of a recursive type. The cons list type we'll define is straightforward: it represents recursion; therefore, the concepts in the example we'll work with will be useful for more complex situations involving recursive types.

More Information About the Cons List

A *cons list* is a data structure that comes from the Lisp programming language. In Lisp, the `cons` function (short for "construct function") constructs a new pair of arguments, which usually are a single value and another pair. These pairs combine to form a list.

The `cons` function concept has made its way into more general functional programming. The expression `cons x onto y` informally means to construct a new container instance by putting the value `x` at the start of this new container, followed by the container `y`.

Each item in a cons list contains two elements: the value of the current item and the next item in the list. The last item in the list contains only a value called `Nil` without a next item. A cons list can be defined recursively by calling the `cons` function. The canonical name to denote the basic cons list is `Nil`. Note that this is not the same as the "null" or "nil" concept in Chapter 1.

Although functional programming languages use cons lists frequently, the `Vec` type is a better choice for a general-purpose list in Rust. Most of the time when you have a list of items in Rust, it's better to use `Vec`. Other, more complex recursive data types *are* useful in functional programming. By starting with the cons list, we can explore how boxes let us define a recursive type without much distraction.

Listing 15-2 contains an enum definition for a cons list. Note that this code won't work because the `List` type doesn't have a known size, which we'll demonstrate.

Filename: `src/main.rs`

```
enum List {
    Cons(i32, List),
    Nil,
}
```

Listing 15-2: The first attempt at defining an enum to represent a cons list data structure. It uses boxes to store the values

Note: We're implementing a cons list that holds only `i32` values for the p example. We could have implemented it using generics, as we discussed i define a cons list type that could store values of any type.

Using the `List` type to store the list `1, 2, 3` would look like the code in Lis

Filename: src/main.rs

```
use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

Listing 15-3: Using the `List` enum to store the list `1, 2, 3`

The first `Cons` value holds `1` and another `List` value. This `List` value is ar holds `2` and another `List` value. This `List` value is one more `Cons` value List value, which is finally `Nil`, the non-recursive variant that signals the e

If we try to compile the code in Listing 15-3, we get the error shown in Listing

```
error[E0072]: recursive type `List` has infinite size
--> src/main.rs:1:1
|
1 | enum List {
| ^^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
|             ----- recursive without indirection
|
= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some
make `List` representable
```

Listing 15-4: The error we get when attempting to define a recursive enum

The error shows this type "has infinite size." The reason is that we've defined that is recursive: it holds another value of itself directly. As a result, Rust can space it needs to store a `List` value. Let's break down why we get this error how Rust decides how much space it needs to store a value of a non-recursive

Computing the Size of a Non-Recursive Type

Recall the `Message` enum we defined in Listing 6-2 when we discussed enum 6:

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

To determine how much space to allocate for a `Message` value, Rust goes th variants to see which variant needs the most space. Rust sees that `Message`: space, `Message::Move` needs enough space to store two `i32` values, and sc variant will be used, the most space a `Message` value will need is the space i largest of its variants.

Contrast this with what happens when Rust tries to determine how much space the `List` enum in Listing 15-2 needs. The compiler starts by looking at the `Cons` variant. It holds a value of type `i32` and a value of type `List`. Therefore, `Cons` needs space equal to the size of an `i32` plus the size of a `List`. To figure out how much space `List` needs, the compiler looks at the variants, starting with the `Cons` variant. The `Cons` variant holds a value of type `i32` and a value of type `List`, and this process continues indefinitely, creating an infinite regress. Figure 15-1 illustrates this infinite list.

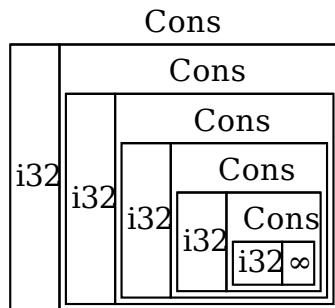


Figure 15-1: An infinite `List` consisting of infinite `cons` variants

Using `Box<T>` to Get a Recursive Type with a Known Size

Rust can't figure out how much space to allocate for recursively defined types, as the error in Listing 15-4. But the error does include this helpful suggestion:

= help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to make `List` representable

In this suggestion, “indirection” means that instead of storing a value directly, store the value in a pointer. Instead of creating a structure to store the value directly, create a structure to store a pointer to the value instead.

Because a `Box<T>` is a pointer, Rust always knows how much space a `Box<T>` occupies, regardless of the amount of data it's pointing to. This means we can use `Box` to store the `Cons` variant instead of another `List` value directly. The `Box<T>` will point to the value that will be on the heap rather than inside the `Cons` variant. Conceptually, this means that lists created with `Box` will “hold” other lists, but this implementation is now more memory efficient than lists created with `Box` that are next to one another rather than inside one another.

We can change the definition of the `List` enum in Listing 15-2 and the usage in Listing 15-3 to the code in Listing 15-5, which will compile:

Filename: src/main.rs

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::*;

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

Listing 15-5: Definition of `List` that uses `Box<T>` in order to have a known size.

The `Cons` variant will need the size of an `i32` plus the space to store the box. The `Nil` variant stores no values, so it needs less space than the `Cons` variant. The `List` value will take up the size of an `i32` plus the size of a box's pointer data. Because the `Box` type can break the infinite, recursive chain, so the compiler can figure out the size of the `List` value. Figure 15-2 shows what the `Cons` variant looks like now.

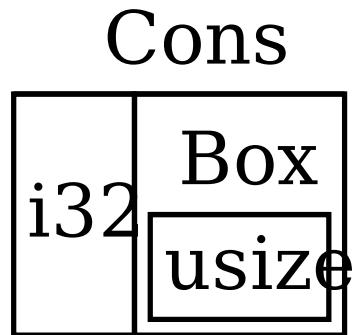


Figure 15-2: A `List` that is not infinitely sized because `Cons` holds a `Box`.

Boxes provide only the indirection and heap allocation; they don't have any other capabilities, like those we'll see with the other smart pointer types. They also incur performance overhead that these special capabilities incur, so they can be used in cases where the indirection is the only feature we need. We'll look at more about boxes in Chapter 17, too.

The `Box<T>` type is a smart pointer because it implements the `Deref` trait, which allows values to be treated like references. When a `Box<T>` value goes out of scope, the heap box it is pointing to is cleaned up as well because of the `Drop` trait implemented by `Box`. We'll implement these two traits in more detail. These two traits will be even more important provided by the other smart pointer types we'll discuss in the rest of this chapter.

Treating Smart Pointers Like Regular References

Deref Trait

Implementing the `Deref` trait allows you to customize the behavior of the `deref` operator (as opposed to the multiplication or glob operator). By implementing `Deref`, a smart pointer can be treated like a regular reference, you can write code that handles references and use that code with smart pointers too.

Let's first look at how the dereference operator works with regular references. We'll define a custom type that behaves like `Box<T>`, and see why the dereference operator works like a reference on our newly defined type. We'll explore how implementing the `Deref` trait is possible for smart pointers to work in a similar way as references. Then we'll look at the `Coerce` feature and how it lets us work with either references or smart pointers.

There's one big difference between the `MyBox<T>` type we're about to build and `Box<T>`: our version will not store its data on the heap. We are focusing the `Deref` trait, and so where the data is actually stored is less important than the

behavior.

Following the Pointer to the Value with the Dereference Operator

A regular reference is a type of pointer, and one way to think of a pointer is it's stored somewhere else. In Listing 15-6, we create a reference to an `i32` value and use the dereference operator to follow the reference to the data:

Filename: `src/main.rs`

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-6: Using the dereference operator to follow a reference to an `i32`

The variable `x` holds an `i32` value, `5`. We set `y` equal to a reference to `x`, which is equal to `5`. However, if we want to make an assertion about the value in `y`, we need to follow the reference to the value it's pointing to (hence *dereference*). Once we have access to the integer value `y` is pointing to that we can compare with `5`.

If we tried to write `assert_eq!(5, y);` instead, we would get this compilation error:

```
error[E0277]: the trait bound `&{integer}: std::cmp::PartialEq<&{integer}` is not satisfied
--> src/main.rs:6:5
 |
6 |     assert_eq!(5, y);
|     ^^^^^^^^^^^^^^ can't compare `&{integer}` with `&{integer}`
|
= help: the trait `std::cmp::PartialEq<&{integer}>` is not implemented for `&{integer}`
```

Comparing a number and a reference to a number isn't allowed because the `assert_eq!` macro must use the dereference operator to follow the reference to the value it's pointing to.

Using `Box<T>` Like a Reference

We can rewrite the code in Listing 15-6 to use a `Box<T>` instead of a reference. The dereference operator will work as shown in Listing 15-7:

Filename: `src/main.rs`

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-7: Using the dereference operator on a `Box<i32>`

The only difference between Listing 15-7 and Listing 15-6 is that here we set `y` to a `Box` containing the value of `x`.

box pointing to the value in `x` rather than a reference pointing to the value assertion, we can use the dereference operator to follow the box's pointer instead when `y` was a reference. Next, we'll explore what is special about `Box<T>` the dereference operator by defining our own box type.

Defining Our Own Smart Pointer

Let's build a smart pointer similar to the `Box<T>` type provided by the standard library. We'll start by defining a `MyBox<T>` type in the same way. Then we'll implement the ability to use the dereference operator.

The `Box<T>` type is ultimately defined as a tuple struct with one element, so we'll define the `MyBox<T>` type in the same way. We'll also define a `new` function to match the standard library's `Box::new`.

Filename: `src/main.rs`

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

Listing 15-8: Defining a `MyBox<T>` type

We define a struct named `MyBox` and declare a generic parameter `T`, because it can hold values of any type. The `MyBox` type is a tuple struct with one element of type `T`. The `MyBox::new` function takes one parameter of type `T` and returns a `MyBox` instance with the value passed in.

Let's try adding the `main` function in Listing 15-7 to Listing 15-8 and changing the code to use the `MyBox<T>` type we've defined instead of `Box<T>`. The code in Listing 15-9 won't compile because the compiler doesn't know how to dereference `MyBox`.

Filename: `src/main.rs`

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 15-9: Attempting to use `MyBox<T>` in the same way we used references

Here's the resulting compilation error:

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |     ^
```

Our `MyBox<T>` type can't be dereferenced because we haven't implemented the `Deref` trait for it. To enable dereferencing with the `*` operator, we implement the `Deref` trait for `MyBox`.

Treating a Type Like a Reference by Implementing the `Deref`

As discussed in Chapter 10, to implement a trait, we need to provide implementation required methods. The `Deref` trait, provided by the standard library, requires a method named `deref` that borrows `self` and returns a reference to the inner type. If our type contains an implementation of `Deref` to add to the definition of `MyBox`:

Filename: src/main.rs

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Listing 15-10: Implementing `Deref` on `MyBox<T>`

The `type Target = T;` syntax defines an associated type for the `Deref` trait. Associated types are a slightly different way of declaring a generic parameter, but you don't need to worry about them for now; we'll cover them in more detail in Chapter 19.

We fill in the body of the `deref` method with `&self.0` so `deref` returns a reference that we want to access with the `*` operator. The `main` function in Listing 15-9 that calls `*y` now compiles, and the assertions pass!

Without the `Deref` trait, the compiler can only dereference `&` references. This gives the compiler the ability to take a value of any type that implements `Deref` and implement a method to get a `&` reference that it knows how to dereference.

When we entered `*y` in Listing 15-9, behind the scenes Rust actually ran this:

```
*(y.deref())
```

Rust substitutes the `*` operator with a call to the `deref` method and then performs the dereference. This means we don't have to think about whether or not we need to call the `deref` method ourselves; we can just write code that functions identically whether we have a regular reference or a reference that implements `Deref`.

The reason the `deref` method returns a reference to a value and that the parentheses in `*(y.deref())` are still necessary is the ownership system. If the `deref` method returned the value directly instead of a reference to the value, the value would be moved out of `y` and into `self`. We don't want to take ownership of the inner value inside `MyBox<T>` in cases where we use the dereference operator.

Note that the `*` operator is replaced with a call to the `deref` method and then the dereference operator just once, each time we use a `*` in our code. Because the substitution does not recurse infinitely, we end up with data of type `i32`, which matches the type of the value in Listing 15-9.

Implicit Deref Coercions with Functions and Methods

Deref coercion is a convenience that Rust performs on arguments to functions. Deref coercion converts a reference to a type that implements `Deref` into a reference to the type itself.

`Deref` can convert the original type into. Deref coercion happens automatically to a particular type's value as an argument to a function or method parameter type in the function or method definition. A sequence of calls to `deref` converts the type we provided into the type the parameter needs.

Deref coercion was added to Rust so that programmers writing functions don't need to add as many explicit references and dereferences with `&` and `*`. This also lets us write more code that can work for either references or smart pointers.

To see deref coercion in action, let's use the `MyBox<T>` type we defined in Listing 15-10. We'll implement the `Deref` trait on `MyBox<String>` that we added in Listing 15-10. Listing 15-11 shows the `hello` function that has a string slice parameter:

Filename: src/main.rs

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

Listing 15-11: A `hello` function that has the parameter `name` of type `&str`

We can call the `hello` function with a string slice as an argument, such as `hello("Rust")`. For example, Deref coercion makes it possible to call `hello` with a reference to a `MyBox<String>`, as shown in Listing 15-12:

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Listing 15-12: Calling `hello` with a reference to a `MyBox<String>` value, which uses Deref coercion

Here we're calling the `hello` function with the argument `&m`, which is a reference to a `MyBox<String>` value. Because we implemented the `Deref` trait on `MyBox<String>`, we can turn `&MyBox<String>` into `&String` by calling `deref`. The standard library provides an implementation of `Deref` on `String` that returns a string slice, and this is implemented for `MyBox<String>`. Rust calls `deref` again to turn the `&String` into `&str`, which matches the `hello` function's definition.

If Rust didn't implement deref coercion, we would have to write the code in Listing 15-12 like the code in Listing 15-13 to call `hello` with a value of type `&MyBox<String>`.

Filename: src/main.rs

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

Listing 15-13: The code we would have to write if Rust didn't have deref coercion

The `(&(*m)[..])` dereferences the `MyBox<String>` into a `String`. Then the `&` and `[..]` are used to get a slice of the `String` that is equal to the whole string to match the signature of `hello`. Writing all of these coercions is harder to read, write, and understand than using Deref coercion. Deref coercion allows Rust to handle these conversions for us automatically.

When the `Deref` trait is defined for the types involved, Rust will analyze the `Deref::deref` as many times as necessary to get a reference to match the pattern. The number of times that `Deref::deref` needs to be inserted is resolved at compile time, so there is no runtime penalty for taking advantage of deref coercion!

How Deref Coercion Interacts with Mutability

Similar to how you use the `Deref` trait to override the `*` operator on immutable references, you can use the `DerefMut` trait to override the `*` operator on mutable references.

Rust does deref coercion when it finds types and trait implementations in the same scope.

- From `&T` to `&U` when `T: Deref<Target=U>`
- From `&mut T` to `&mut U` when `T: DerefMut<Target=U>`
- From `&mut T` to `&U` when `T: Deref<Target=U>`

The first two cases are the same except for mutability. The first case states that if `T` and `U` implement `Deref` to some type `U`, you can get a `&U` transparently. The third case is similar, but it's important to note that the same deref coercion happens for mutable references.

The third case is trickier: Rust will also coerce a mutable reference to an immutable one. This reverse coercion is *not* possible: immutable references will never coerce to mutable references. This is because of the borrowing rules, if you have a mutable reference, that mutable reference must be the only reference to that data (otherwise, the program wouldn't compile). Converting a mutable reference to one immutable reference will never break the borrowing rules. Converting an immutable reference to a mutable reference would require that there is only one immutable reference to that data, and the borrowing rules don't guarantee that. Therefore, Rust can't make this conversion, so converting an immutable reference to a mutable reference is possible.

Running Code on Cleanup with the `Drop` Trait

The second trait important to the smart pointer pattern is `Drop`, which lets you run code when a value is about to go out of scope. You can provide an implementation of this trait on any type, and the code you specify can be used to release resources or clean up connections. We're introducing `Drop` in the context of smart pointers because the `Drop` trait is almost always used when implementing a smart pointer. For example, when you drop a `Box`, it customizes `Drop` to deallocate the space on the heap that the box points to.

In some languages, the programmer must call code to free memory or resolve file handles when they are finished using an instance of a smart pointer. If they forget, the system might crash. In Rust, you can specify that a particular bit of code be run whenever a value goes out of scope, and the compiler will insert this code automatically. As a result, you don't need to worry about placing cleanup code everywhere in a program that uses smart pointers—you still won't leak resources!

Specify the code to run when a value goes out of scope by implementing the `Drop` trait. This trait requires you to implement one method named `drop` that takes a mutable reference to the type. To see when Rust calls `drop`, let's implement `drop` with `println!` statements.

Listing 15-14 shows a `CustomSmartPointer` struct whose only custom function is `drop`. It drops the `CustomSmartPointer!` when the instance goes out of scope. This is printed to the terminal when Rust runs the `drop` function.

Filename: `src/main.rs`

```

struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer { data: String::from("my stuff") };
    let d = CustomSmartPointer { data: String::from("other stuff") };
    println!("CustomSmartPointers created.");
}

```

Listing 15-14: A `CustomSmartPointer` struct that implements the `Drop` trait cleanup code

The `Drop` trait is included in the prelude, so we don't need to import it. We implement the `Drop` trait on `CustomSmartPointer` and provide an implementation for the `drop` method. The body of the `drop` function is where you would place any logic that needs to run when an instance of your type goes out of scope. We're printing some text here because when Rust will call `drop`.

In `main`, we create two instances of `CustomSmartPointer` and then print "CustomSmartPointers created.". At the end of `main`, our instances of `CustomSmartPointer` go out of scope, and Rust will call the code we put in the `drop` method, printing the message "Dropping CustomSmartPointer with data 'other stuff'". Note that we didn't need to call the `drop` method explicitly.

When we run this program, we'll see the following output:

```

CustomSmartPointers created.
Dropping CustomSmartPointer with data `other stuff`!
Dropping CustomSmartPointer with data `my stuff`!

```

Rust automatically called `drop` for us when our instances went out of scope as specified. Variables are dropped in the reverse order of their creation, so `d` is dropped before `c`. This example gives you a visual guide to how the `drop` method works; usually, you'll want to add cleanup code that your type needs to run rather than a print message.

Dropping a Value Early with `std::mem::drop`

Unfortunately, it's not straightforward to disable the automatic `drop` functionality. While it's not usually necessary, the whole point of the `Drop` trait is that it's taken care of automatically. Occasionally, however, you might want to clean up a value early. One example is when you have pointers that manage locks: you might want to force the `drop` method to run earlier than the rest of the code in the same scope. Rust doesn't let you call the `drop` method manually; instead you have to call the `std::mem::drop` function provided by the standard library if you want to force a value to be dropped before the end of its scope.

If we try to call the `Drop` trait's `drop` method manually by modifying the code in Listing 15-14, as shown in Listing 15-15, we'll get a compiler error:

Filename: `src/main.rs`

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Listing 15-15: Attempting to call the `drop` method from the `Drop` trait manually

When we try to compile this code, we'll get this error:

```
error[E0040]: explicit use of destructor method
--> src/main.rs:14:7
 |
14 |     c.drop();
   |     ^^^^^ explicit destructor calls not allowed
```

This error message states that we're not allowed to explicitly call `drop`. The term *destructor*, which is the general programming term for a function that destroys an instance, is analogous to a *constructor*, which creates an instance. The `drop` method is a particular destructor.

Rust doesn't let us call `drop` explicitly because Rust would still automatically drop the value at the end of `main`. This would be a *double free* error because Rust would be trying to free the same memory twice.

We can't disable the automatic insertion of `drop` when a value goes out of scope, but we can call the `drop` method explicitly. So, if we need to force a value to be cleaned up earlier than the end of its scope, we can use the `std::mem::drop` function.

The `std::mem::drop` function is different than the `drop` method in the `Drop` trait because it's a function that takes a value and drops it, instead of passing the value we want to force to be dropped early as an argument. The `drop` method is part of the `Drop` trait, so we can modify `main` in Listing 15-15 to call the `drop` function, a function provided by the `std::mem` module.

Filename: `src/main.rs`

```
fn main() {
    let c = CustomSmartPointer { data: String::from("some data") };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Listing 15-16: Calling `std::mem::drop` to explicitly drop a value before it goes out of scope

Running this code will print the following:

```
CustomSmartPointer created.
Dropping CustomSmartPointer with data `some data`!
CustomSmartPointer dropped before the end of main.
```

The text `Dropping CustomSmartPointer with data `some data`!` is printed before the text `CustomSmartPointer created.` and `CustomSmartPointer dropped before the end of main.`, showing that the `drop` method code is called to drop `c` at that point.

You can use code specified in a `Drop` trait implementation in many ways to make your code convenient and safe: for instance, you could use it to create your own memory management system. Because the `Drop` trait and Rust's ownership system, you don't have to remember to clean up memory manually.

You also don't have to worry about problems resulting from accidentally cleaning up memory multiple times.

use: the ownership system that makes sure references are always valid also called only once when the value is no longer being used.

Now that we've examined `Box<T>` and some of the characteristics of smart pointers, let's look at a few other smart pointers defined in the standard library.

`Rc<T>`, the Reference Counted Smart Pointer

In the majority of cases, ownership is clear: you know exactly which variable owns a value. However, there are cases when a single value might have multiple owners. For example, in linked data structures, multiple edges might point to the same node, and that node should only be cleaned up when all of the edges that point to it are removed. A node shouldn't be cleaned up unless it can't be reached by any edge pointing to it.

To enable multiple ownership, Rust has a type called `Rc<T>`, which is an abbreviation for "reference counting". The `Rc<T>` type keeps track of the number of references to a value. If the reference count reaches zero, whether or not a value is still in use. If there are zero references to a value, the value is deallocated without any references becoming invalid.

Imagine `Rc<T>` as a TV in a family room. When one person enters to watch the TV, they become a reference to it. Others can come into the room and watch the TV. When the last person leaves the room, the TV is turned off because it's no longer being used. If someone turns off the TV while others are still watching it, there would be uproar from the remaining TV watchers!

We use the `Rc<T>` type when we want to allocate some data on the heap for a program to read and we can't determine at compile time which part will finish first. For example, if we knew which part would finish last, we could just make that part the data's owner and the ownership rules enforced at compile time would take effect.

Note that `Rc<T>` is only for use in single-threaded scenarios. When we discuss multithreading in Chapter 16, we'll cover how to do reference counting in multithreaded programs.

Using `Rc<T>` to Share Data

Let's return to our cons list example in Listing 15-5. Recall that we defined it so that we'll create two lists that both share ownership of a third list. Conceptually, this is shown in Figure 15-3:

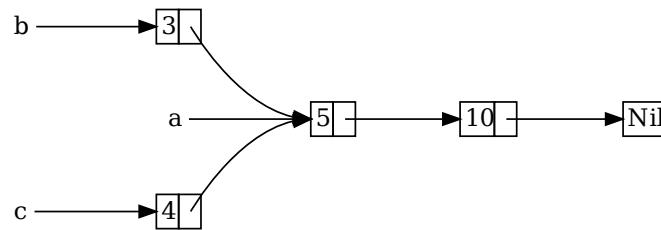


Figure 15-3: Two lists, `b` and `c`, sharing ownership of a third list, `a`

We'll create list `a` that contains `5` and then `10`. Then we'll make two more lists, `b` and `c`, that both share ownership of list `a`. Both `b` and `c` lists will then continue on to the first node after `10`. In other words, both lists will share the first list containing `5` and `10`.

Trying to implement this scenario using our definition of `List` with `Box<T>`

Listing 15-17:

Filename: src/main.rs

```

enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}

```

Listing 15-17: Demonstrating we're not allowed to have two lists using `Box` ownership of a third list

When we compile this code, we get this error:

```

error[E0382]: use of moved value: `a`
--> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
   |                           - value moved here
13 |     let c = Cons(4, Box::new(a));
   |                           ^ value used here after move
   |
= note: move occurs because `a` has type `List`, which does not
       have the `Copy` trait

```

The `Cons` variants own the data they hold, so when we create the `b` list, `a` owns `a`. Then, when we try to use `a` again when creating `c`, we're not allowed to do so because `a` has been moved.

We could change the definition of `Cons` to hold references instead, but then we would need to specify lifetime parameters. By specifying lifetime parameters, we would be able to ensure that the element in the list will live at least as long as the entire list. The borrow checker would catch the error if we tried to compile `let a = Cons(10, &Nil);` for example, because the temporary `Nil` value would be dropped before `a` could take a reference to it.

Instead, we'll change our definition of `List` to use `Rc<T>` in place of `Box<T>`. Listing 15-18. Each `Cons` variant will now hold a value and an `Rc<List>` pointing to a list. Instead of taking ownership of `a`, we'll clone the `Rc<List>` that `a` is holding. This means that the number of references from one to two and letting `a` and `b` share ownership of the same `Rc<List>`. We'll also clone `a` when creating `c`, increasing the number of references to three. Every time we call `Rc::clone`, the reference count to the data within the `Rc` increases by one. The data won't be cleaned up unless there are zero references left.

Filename: src/main.rs

```

enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}

```

Listing 15-18: A definition of `List` that uses `Rc<T>`

We need to add a `use` statement to bring `Rc<T>` into scope because it's not we create the list holding 5 and 10 and store it in a new `Rc<List>` in `a`. The and `c`, we call the `Rc::clone` function and pass a reference to the `Rc<List`

We could have called `a.clone()` rather than `Rc::clone(&a)`, but Rust's cor `Rc::clone` in this case. The implementation of `Rc::clone` doesn't make a c like most types' implementations of `clone` do. The call to `Rc::clone` only ir count, which doesn't take much time. Deep copies of data can take a lot of ti for reference counting, we can visually distinguish between the deep-copy ki kinds of clones that increase the reference count. When looking for perform code, we only need to consider the deep-copy clones and can disregard calls

Cloning an `Rc<T>` Increases the Reference Count

Let's change our working example in Listing 15-18 so we can see the referen we create and drop references to the `Rc<List>` in `a`.

In Listing 15-19, we'll change `main` so it has an inner scope around list `c`; th reference count changes when `c` goes out of scope.

Filename: `src/main.rs`

```

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&c));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}

```

Listing 15-19: Printing the reference count

At each point in the program where the reference count changes, we print t which we can get by calling the `Rc::strong_count` function. This function is rather than `count` because the `Rc<T>` type also has a `weak_count`; we'll see used for in the "Preventing Reference Cycles" section.

This code prints the following:

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

We can see that the `Rc<List>` in `a` has an initial reference count of 1; then, the count goes up by 1. When `c` goes out of scope, the count goes down by 1. To implement a function to decrease the reference count like we have to call `Rc::clone` to increase the count: the implementation of the `Drop` trait decreases the reference count if the `Rc<T>` value goes out of scope.

What we can't see in this example is that when `b` and then `a` go out of scope, the count is then 0, and the `Rc<List>` is cleaned up completely at that point. This allows a single value to have multiple owners, and the count ensures that the value remains valid as long as any of the owners still exist.

Via immutable references, `Rc<T>` allows you to share data between multiple threads for reading only. If `Rc<T>` allowed you to have multiple mutable references simultaneously, it would violate one of the borrowing rules discussed in Chapter 4: multiple mutable borrow on the same data at the same time would cause data races and inconsistencies. But being able to mutate data is very useful, so in the next section, we'll discuss the interior mutability pattern and the `RefCell<T>` type, which allows you to work with this immutability restriction.

RefCell<T> and the Interior Mutability Pattern

Interior mutability is a design pattern in Rust that allows you to mutate data even if there are immutable references to that data; normally, this action is disallowed by the language. To allow mutation while holding immutable references, the pattern uses `unsafe` code inside a data structure to bend the rules of borrowing. We haven't yet covered unsafe code; we will do so in Chapter 14. For now, we'll use types that use the interior mutability pattern when we can ensure that they will be followed at runtime, even though the compiler can't guarantee that. The `RefCell` type is safe because it is wrapped in a safe API, and the outer type is still immutable.

Let's explore this concept by looking at the `RefCell<T>` type that follows the interior mutability pattern.

Enforcing Borrowing Rules at Runtime with RefCell<T>

Unlike `Rc<T>`, the `RefCell<T>` type represents single ownership over the data. This makes `RefCell<T>` different from a type like `Box<T>`? Recall the borrowing rules from Chapter 4:

- At any given time, you can have *either* (but not both of) one mutable reference or multiple immutable references.
- References must always be valid.

With references and `Box<T>`, the borrowing rules' invariants are enforced at compile time. With `RefCell<T>`, these invariants are enforced *at runtime*. With references, if you break the rules, you'll get a compiler error. With `RefCell<T>`, if you break these rules, your program will panic at runtime.

The advantages of checking the borrowing rules at compile time are that errors are found sooner in the development process, and there is no impact on runtime performance analysis. The disadvantages are that the borrow checker needs to analyze the code to determine if the rules are violated, and there is a performance overhead for runtime checks.

is the best choice in the majority of cases, which is why this is Rust's default.

The advantage of checking the borrowing rules at runtime instead is that certain scenarios are then allowed, whereas they are disallowed by the compile-time checker, like the Rust compiler, is inherently conservative. Some properties of code are impossible to check at compile time, so the compiler can't guarantee them by analyzing the code: the most famous example is the Halting Problem, which is beyond the scope of this book but is an interesting topic to research.

Because some analysis is impossible, if the Rust compiler can't be sure the code follows ownership rules, it might reject a correct program; in this way, it's conservative. If the Rust compiler rejects a correct program, users wouldn't be able to trust in the guarantees Rust makes. While the compiler rejects a correct program, the programmer will be inconvenienced, but nothing bad will happen. The `RefCell<T>` type is useful when you're sure your code follows the ownership rules, but the compiler is unable to understand and guarantee that.

Similar to `Rc<T>`, `RefCell<T>` is only for use in single-threaded scenarios and will result in a compile-time error if you try using it in a multithreaded context. We'll talk about how to use `RefCell<T>` in a multithreaded program in Chapter 16.

Here is a recap of the reasons to choose `Box<T>`, `Rc<T>`, or `RefCell<T>`:

- `Rc<T>` enables multiple owners of the same data; `Box<T>` and `RefCell<T>` do not.
- `Box<T>` allows immutable or mutable borrows checked at compile time; `Rc<T>` allows immutable borrows checked at compile time; `RefCell<T>` allows immutable borrows checked at runtime.
- Because `RefCell<T>` allows mutable borrows checked at runtime, you can mutate the value inside the `RefCell<T>` even when the `RefCell<T>` is immutable.

Mutating the value inside an immutable value is the *interior mutability* pattern. In this section, we'll look at situations in which interior mutability is useful and examine how it's possible.

Interior Mutability: A Mutable Borrow to an Immutable Value

A consequence of the borrowing rules is that when you have an immutable variable, you can't borrow it as mutable. For example, this code won't compile:

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

If you tried to compile this code, you'd get the following error:

```
error[E0596]: cannot borrow immutable local variable `x` as mutable
--> src/main.rs:3:18
 |
2 |     let x = 5;
|         - consider changing this to `mut x`
3 |     let y = &mut x;
|             ^ cannot borrow mutably
```

However, there are situations in which it would be useful for a value to mutably borrow an immutable variable. Code outside the value's methods won't be able to change the value. Using `RefCell<T>` is one way to get the ability to have interior mutability. While `RefCell<T>` doesn't get around the borrowing rules completely: the borrow is checked at runtime. If the code violates the rules, you'll get a `panic!` instead of a compiler error.

Let's work through a practical example where we can use `RefCell<T>` to modify a value and see why that is useful.

A Use Case for Interior Mutability: Mock Objects

A *test double* is the general programming concept for a type used in place of a real object during testing. *Mock objects* are specific types of test doubles that record what happened to them so that you can assert that the correct actions took place.

Rust doesn't have objects in the same sense as other languages have objects, but it does have mock object functionality built into the standard library as some other languages do. You can definitely create a struct that will serve the same purposes as a mock object.

Here's the scenario we'll test: we'll create a library that tracks a value against a quota. It sends messages based on how close to the maximum value the current value is. This could be used to keep track of a user's quota for the number of API calls they're allowed to make. Example 15-20 shows the library code:

Our library will only provide the functionality of tracking how close to the maximum value the messages should be at what times. Applications that use our library will provide the mechanism for sending the messages: the application could put a message in a queue, send an email, send a text message, or something else. The library itself doesn't care about that detail. All it needs is something that implements a trait we'll provide called `Messenger`. Example 15-20 shows the library code:

Filename: `src/lib.rs`

```
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 0.75 && percentage_of_max < 0.9 {
            self.messenger.send("Warning: You've used up over 75% of your quota!");
        } else if percentage_of_max >= 0.9 && percentage_of_max < 1.0 {
            self.messenger.send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        }
    }
}
```

Listing 15-20: A library to keep track of how close a value is to a maximum value is at certain levels

One important part of this code is that the `Messenger` trait has one method that takes an immutable reference to `self` and the text of the message. This is the intent that we have. The other important part is that we want to test the behavior of the `set_value` method on the `LimitTracker`. We can change what we pass in for the `value` parameter so that `set_value` doesn't return anything for us to make assertions on. We want to be able to check if the `LimitTracker` has been updated. To do this, we create a `LimitTracker` with something that implements the `Messenger` trait. When we pass different numbers for `value`, the messenger implementation will send appropriate messages.

We need a mock object that, instead of sending an email or text message when we call `send`, only keeps track of the messages it's told to send. We can create a new `MockMessenger` struct that uses the mock object, call the `set_value` method on it, and then check that the mock object has the messages we expect. Listing 15-21 shows how to implement a mock object to do just that, but the borrow checker won't allow us to do it quite like we want to.

Filename: `src/lib.rs`

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

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: vec![] }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

Listing 15-21: An attempt to implement a `MockMessenger` that isn't allowed by the borrow checker

This test code defines a `MockMessenger` struct that has a `sent_messages` field to store values to keep track of the messages it's told to send. We also define an associated function `new` that creates a new `MockMessenger` value that starts with an empty list of messages. We then implement the `Messenger` trait for `MockMessenger` so we can give it to the `LimitTracker`. In the definition of the `send` method, we take the message parameter by value and store it in the `MockMessenger` list of `sent_messages`.

In the test, we're testing what happens when the `LimitTracker` is told to send a message.

that is more than 75 percent of the `max` value. First, we create a new `MockMessenger` and start with an empty list of messages. Then we create a new `LimitTracker` and pass it the new `MockMessenger` and a `max` value of 100. We call the `set_value` method on the `LimitTracker` with a value of 80, which is more than 75 percent of 100. The `LimitTracker` tracks the number of messages that the `MockMessenger` is keeping track of and should now have or

However, there's one problem with this test, as shown here:

```
error[E0596]: cannot borrow immutable field `self.sent_messages` as mutable
--> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str) {
   |             ^---- use `&mut self` here to make mutable
52 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable
```

We can't modify the `MockMessenger` to keep track of the messages, because we have an immutable reference to `self`. We also can't take the suggestion from the error message and use `&mut self` instead, because then the signature of `send` wouldn't match the `Messenger` trait definition (feel free to try and see what error message you get).

This is a situation in which interior mutability can help! We'll store the `sent_messages` field in a `RefCell<Vec<String>>`, and then the `send` message will be able to modify `sent_messages`, which is the same `Vec<String>` we've seen. Listing 15-22 shows what that looks like:

Filename: `src/lib.rs`

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

    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--
        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

Listing 15-22: Using `RefCell<T>` to mutate an inner value while the outer value is immutable

The `sent_messages` field is now of type `RefCell<Vec<String>>` instead of `Vec<String>`. In the `send` function, we create a new `RefCell<Vec<String>>` instance around the empty vector and push the message onto that.

For the implementation of the `send` method, the first parameter is still an `ir self`, which matches the trait definition. We call `borrow_mut` on the `RefCell self.sent_messages` to get a mutable reference to the value inside the `Ref` which is the vector. Then we can call `push` on the mutable reference to the messages sent during the test.

The last change we have to make is in the assertion: to see how many items we call `borrow` on the `RefCell<Vec<String>>` to get an immutable reference.

Now that you've seen how to use `RefCell<T>`, let's dig into how it works!

Keeping Track of Borrows at Runtime with `RefCell<T>`

When creating immutable and mutable references, we use the `&` and `&mut`. With `RefCell<T>`, we use the `borrow` and `borrow_mut` methods, which are belongs to `RefCell<T>`. The `borrow` method returns the smart pointer type `borrow_mut` returns the smart pointer type `RefMut<T>`. Both types implement `Deref` and `DerefMut`, so we can treat them like regular references.

The `RefCell<T>` keeps track of how many `Ref<T>` and `RefMut<T>` smart pointers are active. Every time we call `borrow`, the `RefCell<T>` increases its count of how many borrows are active. When a `Ref<T>` value goes out of scope, the count of immutables decreases by one. When a `RefMut<T>` value goes out of scope, the count of mutables decreases by one. Just like the compile-time borrowing rules, `RefCell<T>` lets us have multiple borrows or one mutable borrow at any point in time.

If we try to violate these rules, rather than getting a compiler error as we would with the compile-time rules, the implementation of `RefCell<T>` will panic at runtime. Listing 15-23 shows a simple example of the implementation of `send` in Listing 15-22. We're deliberately trying to create two mutable borrows in the same scope to illustrate that `RefCell<T>` prevents us from doing that.

Filename: `src/lib.rs`

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

Listing 15-23: Creating two mutable references in the same scope to see that `RefCell` prevents us from doing that.

We create a variable `one_borrow` for the `RefMut<T>` smart pointer returned by `borrow_mut`. Then we create another mutable borrow in the same way in the variable `two_borrow`. This violates the rule that we can't have two mutable references in the same scope, which isn't allowed. When we run the code in Listing 15-23, the compiler will complain:

```
---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at
'already borrowed: BorrowMutError', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Notice that the code panicked with the message `already borrowed: BorrowMutError`. `RefCell<T>` handles violations of the borrowing rules at runtime.

Catching borrowing errors at runtime rather than compile time means that you won't find out about your bugs until you run your code later in the development process and possibly not until your code has shipped.

production. Also, your code would incur a small runtime performance penalty of tracking the borrows at runtime rather than compile time. However, using `Rc` is possible to write a mock object that can modify itself to keep track of the mutable values you're using it in a context where only immutable values are allowed. You can despite its trade-offs to get more functionality than regular references provided by `RefCell`.

Having Multiple Owners of Mutable Data by Combining `Rc<T>`

A common way to use `RefCell<T>` is in combination with `Rc<T>`. Recall that `Rc<T>` gives multiple owners of some data, but it only gives immutable access to that data. If you hold a `RefCell<T>`, you can get a value that can have multiple owners and mutate!

For example, recall the cons list example in Listing 15-18 where we used `Rc<List>` to share ownership of another list. Because `Rc<T>` holds only immutable references, we can't change any of the values in the list once we've created them. Let's add in `Rc`'s ability to change the values in the lists. Listing 15-24 shows that by using a `Rc` definition, we can modify the value stored in all the lists:

Filename: `src/main.rs`

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

Listing 15-24: Using `Rc<RefCell<i32>>` to create a `List` that we can mutate

We create a value that is an instance of `Rc<RefCell<i32>>` and store it in a variable `value` so we can access it directly later. Then we create a `List` in `a` with a `Cons` value. We need to clone `value` so both `a` and `value` have ownership of the inner value, thus transferring ownership from `value` to `a` or having a borrow from `value`.

We wrap the list `a` in an `Rc<T>` so when we create lists `b` and `c`, they can link to what we did in Listing 15-18.

After we've created the lists in `a`, `b`, and `c`, we add 10 to the value in `value` via the `borrow_mut` on `value`, which uses the automatic dereferencing feature we learned about (see the section "Where's the `->` Operator?") to dereference the `Rc<T>` to the inner value. The `borrow_mut` method returns a `RefMut<T>` smart pointer, and we can use it to change the value.

operator on it and change the inner value.

When we print `a`, `b`, and `c`, we can see that they all have the modified value:

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

This technique is pretty neat! By using `RefCell<T>`, we have an outwardly immutable type, but we can use the methods on `RefCell<T>` that provide access to its interior value so we can modify our data when we need to. The runtime checks of the borrowing rules still prevent races, and it's sometimes worth trading a bit of speed for this flexibility in our code.

The standard library has other types that provide interior mutability, such as `Cell` and `Mutex`. These are similar except that instead of giving references to the inner value, the value is copied into the `Cell<T>`. There's also `Mutex<T>`, which offers interior mutability that's safe across multiple threads; we'll discuss its use in Chapter 16. Check out the standard library documentation for more information on the differences between these types.

Reference Cycles Can Leak Memory

Rust's memory safety guarantees make it difficult, but not impossible, to accidentally create a reference cycle that is never cleaned up (known as a *memory leak*). Preventing memory leaks is just as important as Rust's guarantees in the same way that disallowing data races at compile time is important for safety. We can see that Rust allows memory leaks by using `RefCell<T>`: it's possible to create references where items refer to each other in a cycle, and these cycles will never be broken because the reference count of each item in the cycle will never drop to zero; values will never be dropped.

Creating a Reference Cycle

Let's look at how a reference cycle might happen and how to prevent it, starting with the `List` enum and a `tail` method in Listing 15-25:

Filename: `src/main.rs`

```
use std::rc::Rc;
use std::cell::RefCell;
use List::{Cons, Nil};

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match *self {
            Cons(_, ref item) => Some(item),
            Nil => None,
        }
    }
}
```

Listing 15-25: A cons list definition that holds a `RefCell<T>` so we can modify the list while referring to it.

We're using another variation of the `List` definition in Listing 15-5. The second variant is now `RefCell<Rc<List>>`, meaning that instead of having the ability to change the value as we did in Listing 15-24, we want to modify which `List` value a `Cons` node contains. We're also adding a `tail` method to make it convenient for us to access the next item in the list.

In Listing 15-26, we're adding a `main` function that uses the definitions in Listing 15-25. It creates a list in `a` and a list in `b` that points to the list in `a`. Then it modifies `b`, creating a reference cycle. There are `println!` statements along the way to show what the reference counts are at various points in this process.

Filename: `src/main.rs`

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

Listing 15-26: Creating a reference cycle of two `List` values pointing to each other.

We create an `Rc<List>` instance holding a `List` value in the variable `a` with a value of 5. We then create an `Rc<List>` instance holding another `List` value in the variable `b` with a value of 10 and points to the list in `a`.

We modify `a` so it points to `b` instead of `Nil`, creating a cycle. We do that by using the `borrow_mut` method to get a reference to the `RefCell<Rc<List>>` in `a`, which we put in `b`. We then use the `borrow_mut` method on the `RefCell<Rc<List>>` to change the value of the `Rc<List>` that holds a `Nil` value to the `Rc<List>` in `b`.

When we run this code, keeping the last `println!` commented out for the reference counts, the output is:

```
a initial rc count = 1
a next item = Some(RefCell { value: Nil })
a rc count after b creation = 2
b initial rc count = 1
b next item = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b rc count after changing a = 2
a rc count after changing a = 2
```

The reference count of the `Rc<List>` instances in both `a` and `b` are 2 after we've modified `a` to point to `b`. At the end of `main`, Rust will try to drop `b` first, which will decrement the reference counts of the `Rc<List>` instances in `a` and `b` by 1.

However, because `a` is still referencing the `Rc<List>` that was in `b`, that `Rc` rather than 0, so the memory the `Rc<List>` has on the heap won't be dropped; it just sit there with a count of 1, forever. To visualize this reference cycle, we've Figure 15-4.

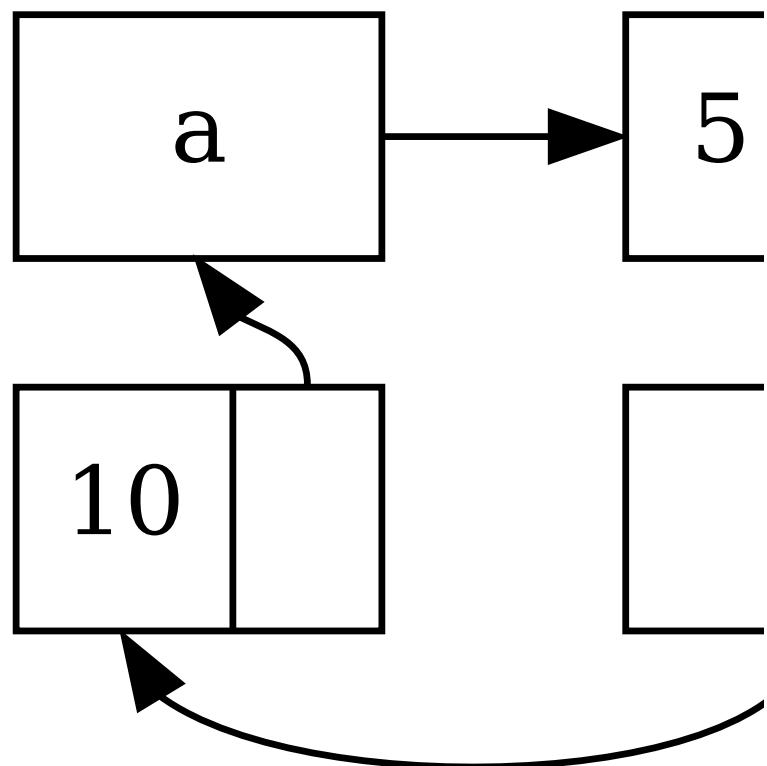


Figure 15-4: A reference cycle of lists `a` and `b` pointing to each other

If you uncomment the last `println!` and run the program, Rust will try to print `a` and `b`, pointing to `b` pointing to `a` and so forth until it overflows the stack.

In this case, right after we create the reference cycle, the program ends. The reference cycles aren't very dire. However, if a more complex program allocated lots of references to a single object and held onto it for a long time, the program would use more memory than it needs, eventually overwhelming the system, causing it to run out of available memory.

Creating reference cycles is not easily done, but it's not impossible either. If you have values that contain `Rc<T>` values or similar nested combinations of types within them, and you're using reference counting, you must ensure that you don't create cycles; you can't break them. Creating a reference cycle would be a logic bug in your program that's hard to find, especially in automated tests, code reviews, and other software development practices to catch.

Another solution for avoiding reference cycles is reorganizing your data structures. Rust's ownership rules express ownership and some references don't. As a result, you can have a mix of some ownership relationships and some non-ownership relationships, and the relationships affect whether or not a value can be dropped. In Listing 15-25, we'll see how to use variants to own their list, so reorganizing the data structure isn't possible. Let's look at how we can use graphs made up of parent nodes and child nodes to see when non-owned references are an appropriate way to prevent reference cycles.

Preventing Reference Cycles: Turning an `Rc<T>` into a `Weak<T>`

So far, we've demonstrated that calling `Rc::clone` increases the `strong_count` instance, and an `Rc<T>` instance is only cleaned up if its `strong_count` is 0. A *weak reference* to the value within an `Rc<T>` instance by calling `Rc::downgrade` reference to the `Rc<T>`. When you call `Rc::downgrade`, you get a smart pointer. Instead of increasing the `strong_count` in the `Rc<T>` instance by 1, calling `downgrade` increases the `weak_count` by 1. The `Rc<T>` type uses `weak_count` to keep track of how many `Weak<T>` references exist, similar to `strong_count`. The difference is the `weak_count` is set to be 0 for the `Rc<T>` instance to be cleaned up.

Strong references are how you can share ownership of an `Rc<T>` instance. They allow you to express an ownership relationship. They won't cause a reference cycle because some weak references will be broken once the strong reference count of value goes to zero.

Because the value that `Weak<T>` references might have been dropped, to determine if a `Weak<T>` is pointing to, you must make sure the value still exists. Do this by calling the `upgrade` method on a `Weak<T>` instance, which will return an `Option<Rc<T>>`. It will return `Some` if the `Rc<T>` value has not been dropped yet and a result of `None` if the value has been dropped. Because `upgrade` returns an `Option<T>`, Rust will ensure that both the `Some` and `None` cases are handled, and there won't be an invalid pointer.

As an example, rather than using a list whose items know only about their parent item, we can use a tree whose items know about their children items *and* their parent items.

Creating a Tree Data Structure: a `Node` with Child `Nodes`

To start, we'll build a tree with nodes that know about their child nodes. We'll define a `Node` that holds its own `i32` value as well as references to its children `Node` instances.

Filename: `src/main.rs`

```
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>,
}
```

We want a `Node` to own its children, and we want to share that ownership with other `Node`s so they can access each `Node` in the tree directly. To do this, we define the `children` item as a `RefCell<Vec<Rc<Node>>`. We also want to modify which nodes are children of another node. Instead of using a plain `Vec`, we use `RefCell<T>` in `children` around the `Vec<Rc<Node>>`.

Next, we'll use our struct definition and create one `Node` instance named `leaf` with the value 0 and no children, and another instance named `branch` with the value 5 and one child, `leaf`, as shown in Listing 15-27:

Filename: `src/main.rs`

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}

```

Listing 15-27: Creating a `leaf` node with no children and a `branch` node with children

We clone the `Rc<Node>` in `leaf` and store that in `branch`, meaning the Node owners: `leaf` and `branch`. We can get from `branch` to `leaf` through `branch.children`, but there's no way to get from `leaf` to `branch`. The reason is that `leaf` has no `parent` field and doesn't know they're related. We want `leaf` to know that `branch` is its parent and `branch` to know that `leaf` is its next.

Adding a Reference from a Child to Its Parent

To make the child node aware of its parent, we need to add a `parent` field to the `Node` definition. The trouble is in deciding what the type of `parent` should be. We could use `Rc<T>`, because that would create a reference cycle with `leaf.parent` pointing to `branch` and `branch.parent` pointing to `leaf`, which would cause their `strong_count` to both grow infinitely.

Thinking about the relationships another way, a parent node should own its child nodes. If a parent node is dropped, its child nodes should be dropped as well. However, a child node should not own its parent: if we drop a child node, the parent should still exist. This is a case for a weak reference.

So instead of `Rc<T>`, we'll make the type of `parent` use `Weak<T>`, specifically `RefCell<Weak<Node>>`. Now our `Node` struct definition looks like this:

Filename: src/main.rs

```

use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

```

A node will be able to refer to its parent node but doesn't own its parent. In Listing 15-27, we modified `main` to use this new definition so the `leaf` node will have a way to refer to its parent.

Filename: src/main.rs

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}

```

Listing 15-28: A `leaf` node with a weak reference to its parent node `branch`

Creating the `leaf` node looks similar to how creating the `leaf` node looked the exception of the `parent` field: `leaf` starts out without a parent, so we create a `Weak<Node>` reference instance.

At this point, when we try to get a reference to the parent of `leaf` by using `leaf.parent`, we get a `None` value. We see this in the output from the first `println!` statement:

```
leaf parent = None
```

When we create the `branch` node, it will also have a new `Weak<Node>` reference because `branch` doesn't have a parent node. We still have `leaf` as one of the children of `branch`. Once we have the `Node` instance in `branch`, we can modify `leaf` to give it a `parent` reference to `branch`. We use the `borrow_mut` method on the `RefCell<Weak<Node>>` cell of `leaf`, and then we use the `Rc::downgrade` function to create a `Weak<Node>` reference to the `Rc<Node>` in `branch`.

When we print the parent of `leaf` again, this time we'll get a `Some` variant because `leaf` now has a parent! When we print `leaf`, we also avoid the cycle that would cause a stack overflow like we had in Listing 15-26; the `Weak<Node>` references are pointers.

```
leaf.parent = Some(Node { value: 5, parent: RefCell { value: (Weak::new(), RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak::new(), RefCell { value: [] }) } ] }) } })
```

The lack of infinite output indicates that this code didn't create a reference cycle. We can verify this by looking at the values we get from calling `Rc::strong_count` and `Rc::weak_count`:

Visualizing Changes to `strong_count` and `weak_count`

Let's look at how the `strong_count` and `weak_count` values of the `Rc<Node>` instances change as we create the tree. We can do this by creating a new inner scope and moving the creation of `branch` into that scope. This way, we can see what happens when `branch` is created and then dropped when it goes out of scope. These modifications are shown in Listing 15-29:

Filename: `src/main.rs`

```

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch strong = {}, weak = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );
    }

    println!(
        "leaf strong = {}, weak = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}

println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
println!(
    "leaf strong = {}, weak = {}",
    Rc::strong_count(&leaf),
    Rc::weak_count(&leaf),
);
}

```

Listing 15-29: Creating `branch` in an inner scope and examining strong and weak counts.

After `leaf` is created, its `Rc<Node>` has a strong count of 1 and a weak count of 0. In the inner scope, we create `branch` and associate it with `leaf`, at which point when we print the counts in `branch`, `Rc<Node>` in `branch` will have a strong count of 1 and a weak count of 1 (for its parent `leaf` in `branch` with a `Weak<Node>`). When we print the counts in `leaf`, we'll see that `leaf` still has a strong count of 1 and a weak count of 0, because `branch` now has a clone of the `Rc<Node>` of `leaf` stored in its `children` field, but `leaf` will still have a weak count of 0.

When the inner scope ends, `branch` goes out of scope and the strong count decreases to 0, so its `Node` is dropped. The weak count of 1 from `leaf.parent.borrow_mut()` is dropped, so we don't get any memory leaks!

If we try to access the parent of `leaf` after the end of the scope, we'll get `None`. At the end of the program, the `Rc<Node>` in `leaf` has a strong count of 1 and a weak count of 0, because the variable `branch` is now the only reference to the `Rc<Node>` again.

All of the logic that manages the counts and value dropping is built into `Rc<T>`'s implementation of the `Drop` trait. By specifying that the relationship from `Node` to `RefCell` should be a `Weak<T>` reference in the definition of `Node`, you're able to have child nodes and vice versa without creating a reference cycle and memory leaks.

Summary

This chapter covered how to use smart pointers to make different guarantees than Rust makes by default with regular references. The `Box<T>` type has a pointer to data allocated on the heap. The `Rc<T>` type keeps track of the number of owners of the heap so that data can have multiple owners. The `RefCell<T>` type with `get_mut` gives us a type that we can use when we need an immutable type but need to mutate that type; it also enforces the borrowing rules at runtime instead of at compile time.

Also discussed were the `Deref` and `Drop` traits, which enable a lot of functionality on smart pointers. We explored reference cycles that can cause memory leaks and how to break them using `Weak<T>`.

If this chapter has piqued your interest and you want to implement your own smart pointer type, check out ["The Rustonomicon"](#) for more useful information.

Next, we'll talk about concurrency in Rust. You'll even learn about a few new concurrency patterns.

Fearless Concurrency

Handling concurrent programming safely and efficiently is another of Rust's strengths. In the past, concurrent programming, where different parts of a program execute independently, and parallelism, where different parts of a program execute at the same time, have been difficult to implement. As more computers take advantage of their multiple processors, the challenge of writing safe concurrent code has become even more difficult. Historically, writing concurrent code has been difficult and error prone: Rust hopes to change that.

Initially, the Rust team thought that ensuring memory safety and preventing deadlocks were two separate challenges to be solved with different methods. Over time, they realized that the ownership and type systems are a powerful set of tools to help manage concurrent and parallel programs. By leveraging ownership and type checking, many concurrency errors are caught at compile time rather than runtime. Therefore, rather than spending lots of time trying to reproduce the exact circumstances under which a runtime error occurs, incorrect code will refuse to compile and present an error explaining what went wrong. Once you understand the error, you can fix your code while you're working on it rather than potential bugs being shipped to production. We've nicknamed this aspect of Rust *fearless concurrency*. Fearless concurrency allows you to write code that is free of subtle bugs and is easy to maintain without introducing new bugs.

Note: For simplicity's sake, we'll refer to many of the problems as *concurrent*. To be more precise by saying *concurrent and/or parallel*. If this book were about parallelism, we'd be more specific. For this chapter, please mentally substitute *concurrent and/or parallel* whenever we use *concurrent*.

Many languages are dogmatic about the solutions they offer for handling concurrency. For example, Erlang has elegant functionality for message-passing concurrency but requires a different way to share state between threads. Supporting only a subset of possible concurrency models is a common strategy for higher-level languages, because a higher-level language promises to take away some control to gain abstractions. However, lower-level languages are expected to support all concurrency models with the best performance in any given situation and have fewer abstractions. Therefore, Rust offers a variety of tools for modeling problems in a way that is appropriate for your situation and requirements.

Here are the topics we'll cover in this chapter:

- How to create threads to run multiple pieces of code at the same time

- *Message-passing* concurrency, where channels send messages between threads.
- *Shared-state* concurrency, where multiple threads have access to some shared state.
- The `Sync` and `Send` traits, which extend Rust's concurrency guarantee to as well as types provided by the standard library

Using Threads to Run Code Simultaneously

In most current operating systems, an executed program's code is run in a *p*rocess. The operating system manages multiple processes at once. Within your program, you can have multiple independent parts that run simultaneously. The features that run these independently are called *threads*.

Splitting the computation in your program into multiple threads can improve performance. While the program does multiple tasks at the same time, but it also adds complexity. If two threads try to run simultaneously, there's no inherent guarantee about the order in which they will run. This can lead to problems, such as:

- Race conditions, where threads are accessing data or resources in an interleaved manner.
- Deadlocks, where two threads are waiting for each other to finish using a shared resource. If one thread has the lock and is waiting, the other thread has, preventing both threads from continuing.
- Bugs that happen only in certain situations and are hard to reproduce.

Rust attempts to mitigate the negative effects of using threads, but programmatic threads still take careful thought and require a code structure that is different from programs running in a single thread.

Programming languages implement threads in a few different ways. Many operating systems provide an API for creating new threads. This model where a language calls the operating system's APIs to create threads is sometimes called *1:1*, meaning one operating system thread per language thread.

Many programming languages provide their own special implementation of threads. Languages that provide threads but do not execute them in the context of a different number of operating system threads are known as *green threads*, and languages that will execute them in the context of a different number of operating system threads are known as *copy-on-write threads*. The green-threaded model is called the *M:N* model: there are M green threads per N operating system threads, where M and N are not necessarily the same number.

Each model has its own advantages and trade-offs, and the trade-off most important is runtime support. *Runtime* is a confusing term and can have different meanings depending on the context.

In this context, by *runtime* we mean code that is included by the language in the binary. The amount of runtime code can be large or small depending on the language, but every non-assembly language includes some amount of runtime code. For that reason, colloquially when people say a language has a "small runtime," they often mean "small runtime." Smaller runtimes have fewer features but larger binaries, resulting in smaller executables, which make it easier to combine the language with other libraries and contexts. Although many languages are okay with increasing the runtime for more features, Rust needs to have nearly no runtime and cannot compromise performance by including C code to maintain performance.

The green-threading M:N model requires a larger language runtime to manage threads. The Rust standard library only provides an implementation of 1:1 threading. Because Rust is a systems-level language, there are crates that implement M:N threading if you would like to use it. These crates provide more control over aspects such as more control over which threads run when and lower-level control over memory management, for example.

Now that we've defined threads in Rust, let's explore how to use the threads in the standard library.

Creating a New Thread with `spawn`

To create a new thread, we call the `thread::spawn` function and pass it a closure (see closures in Chapter 13) containing the code we want to run in the new thread:

Filename: `src/main.rs`

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Listing 16-1: Creating a new thread to print one thing while the main thread

Note that with this function, the new thread will be stopped when the main thread has finished running. The output from this program might be a little different each time you run it, but it will look similar to the following:

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

The calls to `thread::sleep` force a thread to stop its execution for a short duration so that another thread can run. The threads will probably take turns, but that isn't guaranteed by the standard library; how your operating system schedules the threads. In this run, the main thread ran first, though the print statement from the spawned thread appears first in the output. Even though the spawned thread told the main thread to sleep for 1 millisecond, the main thread told the spawned thread to print until `i` is 9, so it only got to 5 before the main thread ended.

If you run this code and only see output from the main thread, or don't see the output from the spawned thread, try increasing the numbers in the ranges to create more opportunities for the operating system to switch between the threads.

Waiting for All Threads to Finish Using `join` Handles

The code in Listing 16-1 not only stops the spawned thread prematurely making the main thread end, but also can't guarantee that the spawned thread will get to run at all! The reason is that there is no guarantee on the order in which threads run!

We can fix the problem of the spawned thread not getting to run, or not getting to run at all, by saving the return value of `thread::spawn` in a variable. The return type of `thread::spawn` is a `JoinHandle`. A `JoinHandle` is an owned value that, when we call the `join` method on it,

for its thread to finish. Listing 16-2 shows how to use the `JoinHandle` of the Listing 16-1 and call `join` to make sure the spawned thread finishes before

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Listing 16-2: Saving a `JoinHandle` from `thread::spawn` to guarantee the thread's completion

Calling `join` on the handle blocks the thread currently running until the thread terminates. *Blocking* a thread means that thread is prevented from proceeding until it exits. Because we've put the call to `join` after the main thread's `for` loop, the output should produce output similar to this:

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

The two threads continue alternating, but the main thread waits because of the `handle.join()` and does not end until the spawned thread is finished.

But let's see what happens when we instead move `handle.join()` before the main thread's `for` loop like this:

Filename: src/main.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

The main thread will wait for the spawned thread to finish and then run its own code. This won't be interleaved anymore, as shown here:

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Small details, such as where `join` is called, can affect whether or not your threads run interleaved.

Using `move` Closures with Threads

The `move` closure is often used alongside `thread::spawn` because it allows the closure to take ownership of the values it uses in the environment. This is especially useful when creating new threads in order to transfer ownership of data from one thread to another.

In Chapter 13, we mentioned we can use the `move` keyword before the parameter name in a closure to force the closure to take ownership of the values it uses in the environment. This is especially useful when creating new threads in order to transfer ownership of data from one thread to another.

Notice in Listing 16-1 that the closure we pass to `thread::spawn` takes no arguments. It's not allowed to use any data from the main thread in the spawned thread's code. To use data from the main thread in the spawned thread, the spawned thread's closure must capture the data from the main thread. Listing 16-3 shows an attempt to create a vector in the main thread and use it in the spawned thread. However, this won't yet work, as you'll see in a moment.

Filename: src/main.rs

```
use std::thread;

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

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Listing 16-3: Attempting to use a vector created by the main thread in another thread.

The closure uses `v`, so it will capture `v` and make it part of the closure's environment. Since `thread::spawn` runs this closure in a new thread, we should be able to access `v` from the closure. But when we compile this example, we get the following error:

```
error[E0373]: closure may outlive the current function, but it borrows `v`, which is owned by the current function
--> src/main.rs:6:32
   |
6 |     let handle = thread::spawn(|| {
   |                         ^^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
   |                         - `v` is borrowed here
   |
   |         help: to force the closure to take ownership of `v` (and any other
   |         variables), use the `move` keyword
   |
6 |     let handle = thread::spawn(move || {
   |                         ^^^^^^
```

Rust *infers* how to capture `v`, and because `println!` only needs a reference to `v`, it borrows `v`. However, there's a problem: Rust can't tell how long the spawned thread will run, so it doesn't know if the reference to `v` will always be valid.

Listing 16-4 provides a scenario that's more likely to have a reference to `v` that drops `v`.

Filename: src/main.rs

```
use std::thread;

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

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

Listing 16-4: A thread with a closure that attempts to capture a reference to `v`, but the main thread drops `v`.

If we were allowed to run this code, there's a possibility the spawned thread would just sit in the background without running at all. The spawned thread has a reference to `v`, but the main thread immediately drops `v`, using the `drop` function we discussed earlier. When the spawned thread starts to execute, `v` is no longer valid, so a reference to it would be invalid! Oh no!

To fix the compiler error in Listing 16-3, we can use the error message's advice:

```
help: to force the closure to take ownership of `v` (and any other
variables), use the `move` keyword
|           let handle = thread::spawn(move || {
|           ^^^^^^
```

By adding the `move` keyword before the closure, we force the closure to take ownership of the values it's using rather than allowing Rust to infer that it should borrow them. The code in Listing 16-3 shown in Listing 16-5 will compile and run as we intend:

Filename: `src/main.rs`

```
use std::thread;

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

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Listing 16-5: Using the `move` keyword to force a closure to take ownership of its arguments

What would happen to the code in Listing 16-4 where the main thread called `drop(v)` after spawning the closure? Would `move` fix that case? Unfortunately, no; we would get a different error. Listing 16-4 is trying to do isn't allowed for a different reason. If we added `move` to the closure in Listing 16-4, the closure would move `v` into the closure's environment, and we could no longer call `drop(v)` from the main thread. We would get this compiler error instead:

```
error[E0382]: use of moved value: `v`
--> src/main.rs:10:10
   |
6 |     let handle = thread::spawn(move || {
   |     ^----- value moved (into closure)
...
10 |     drop(v); // oh no!
   |           ^ value used here after move
   |
   = note: move occurs because `v` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

Rust's ownership rules have saved us again! We got an error from the code in Listing 16-4. The error message says that Rust was being conservative and only borrowing `v` for the thread, which means that the closure could theoretically invalidate the spawned thread's reference. By telling Rust to move `v` to the spawned thread, we're guaranteeing Rust that the main thread won't change `v`. If we changed Listing 16-4 in the same way, we're then violating the ownership rule in the main thread. The `move` keyword overrides Rust's conservative default and lets us violate the ownership rules.

With a basic understanding of threads and the thread API, let's look at what we can do with them.

Using Message Passing to Transfer Data Between Threads

One increasingly popular approach to ensuring safe concurrency is *message passing*.

or actors communicate by sending each other messages containing data. He from the Go language documentation: "Do not communicate by sharing memory by communicating."

One major tool Rust has for accomplishing message-sending concurrency is programming concept that Rust's standard library provides an implementation channel in programming as being like a channel of water, such as a stream carrying something like a rubber duck or boat into a stream, it will travel downstream on a waterway.

A channel in programming has two halves: a transmitter and a receiver. The upstream location where you put rubber ducks into the river, and the receiving end of the rubber duck ends up downstream. One part of your code calls methods on the transmitter to send data you want to send, and another part checks the receiving end for arriving data said to be *closed* if either the transmitter or receiver half is dropped.

Here, we'll work up to a program that has one thread to generate values and another thread to receive them. We'll start with a simple example of a channel, and another thread that will receive the values and print them out. This is a good way to learn how to use channels to pass values between threads using a channel to illustrate the feature. Once you're comfortable with this basic technique, you could use channels to implement a chat system or a system that performs calculations across multiple threads. You could even perform parts of a calculation and send the parts to one thread that aggregates the results.

First, in Listing 16-6, we'll create a channel but not do anything with it. Note that this is a very simple example, and we haven't yet learned how to tell what type of values we want to send over the channel.

Filename: src/main.rs

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

Listing 16-6: Creating a channel and assigning the two halves to `tx` and `rx`

We create a new channel using the `mpsc::channel` function; `mpsc` stands for *multiple producers, single consumer*. In short, the way Rust's standard library implements channels is that they have multiple *sending* ends that produce values but only one *receiving* end that receives values. Imagine multiple streams flowing together into one big river: every stream will end up in one river at the end. We'll start with a single producer and add multiple producers when we get this example working.

The `mpsc::channel` function returns a tuple, the first element of which is the transmitter end and the second element is the receiving end. The abbreviations `tx` and `rx` are traditional abbreviations for *transmitter* and *receiver* respectively, so we name our variables as such. We're using a `let` statement with a pattern that destructures the tuple of patterns in `let` statements and destructuring in Chapter 18. Using a `let` statement is a more convenient approach to extract the pieces of the tuple returned by `mpsc::channel`.

Let's move the transmitting end into a spawned thread and have it send one value over the channel to the main thread. The main thread is communicating with the main thread, as shown in Listing 16-7. This is like putting a rubber duck in the river upstream or sending a chat message from one thread to another.

Filename: src/main.rs

```

use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}

```

Listing 16-7: Moving `tx` to a spawned thread and sending "hi"

Again, we're using `thread::spawn` to create a new thread and then using `move` closure so the spawned thread owns `tx`. The spawned thread needs to own the channel to be able to send messages through the channel.

The transmitting end has a `send` method that takes the value we want to send and returns a `Result<T, E>` type, so if the receiving end has already been dropped to send a value, the send operation will return an error. In this example, we'll panic in case of an error. But in a real application, we would handle it properly to review strategies for proper error handling.

In Listing 16-8, we'll get the value from the receiving end of the channel in the same way like retrieving the rubber duck from the water at the end of the river or like `get`.

Filename: `src/main.rs`

```

use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}

```

Listing 16-8: Receiving the value "hi" in the main thread and printing it

The receiving end of a channel has two useful methods: `recv` and `try_recv`, short for `receive`, which will block the main thread's execution and wait until a value is sent on the channel. Once a value is sent, `recv` will return it in a `Result<T, E>`. When the channel closes, `recv` will return an error to signal that no more values will be sent.

The `try_recv` method doesn't block, but will instead return a `Result<T, E>` value holding a message if one is available and an `Err` value if there aren't any. Using `try_recv` is useful if this thread has other work to do while waiting for a message. We can write a loop that calls `try_recv` every so often, handles a message if one is available, and then continues to do other work for a little while until checking again.

We've used `recv` in this example for simplicity; we don't have any other work to do other than wait for messages, so blocking the main thread is appropriate.

When we run the code in Listing 16-8, we'll see the value printed from the main thread.

```
Got: hi
```

```
Perfect!
```

Channels and Ownership Transference

The ownership rules play a vital role in message sending because they help you write safe concurrent code. Preventing errors in concurrent programming is the advantage of ownership throughout your Rust programs. Let's do an experiment to show how ownership works together to prevent problems: we'll try to use a `val` value in another thread after we've sent it down the channel. Try compiling the code in Listing 16-9 to see what happens:

Filename: `src/main.rs`

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Listing 16-9: Attempting to use `val` after we've sent it down the channel

Here, we try to print `val` after we've sent it down the channel via `tx.send`. This is a bad idea: once the value has been sent to another thread, that thread could modify it. We could then try to use the value again. Potentially, the other thread's modifications could lead to unexpected results due to inconsistent or nonexistent data. However, Rust prevents us from doing this by not letting us compile the code in Listing 16-9:

```
error[E0382]: use of moved value: `val`
--> src/main.rs:10:31
   |
9  |         tx.send(val).unwrap();
   |         ^--- value moved here
10 |         println!("val is {}", val);
   |                     ^^^ value used here after move
   |
   = note: move occurs because `val` has type `std::string::String`
not implement the `Copy` trait
```

Our concurrency mistake has caused a compile time error. The `send` function moves the `val` parameter, and when the value is moved, the receiver takes ownership of it. If we accidentally use the value again after sending it, the ownership system catches us.

Sending Multiple Values and Seeing the Receiver Waiting

The code in Listing 16-8 compiled and ran, but it didn't clearly show us that the two threads were talking to each other over the channel. In Listing 16-10 we've made some changes to make this more apparent.

will prove the code in Listing 16-8 is running concurrently: the spawned thread will receive multiple messages and pause for a second between each message.

Filename: src/main.rs

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Listing 16-10: Sending multiple messages and pausing between each

This time, the spawned thread has a vector of strings that we want to send to the receiver. It iterates over them, sending each individually, and pauses between each by calling `thread::sleep` with a `Duration` value of 1 second.

In the main thread, we're not calling the `recv` function explicitly anymore: it receives `rx` as an iterator. For each value received, we're printing it. When the channel ends, the loop exits.

When running the code in Listing 16-10, you should see the following output printed in between each line:

```
Got: hi
Got: from
Got: the
Got: thread
```

Because we don't have any code that pauses or delays in the `for` loop in the main thread, we can tell that the main thread is waiting to receive values from the spawned thread.

Creating Multiple Producers by Cloning the Transmitter

Earlier we mentioned that `mpsc` was an acronym for *multiple producer, single consumer*. We can use `mpsc` to use and expand the code in Listing 16-10 to create multiple threads that all send to the same receiver. We can do so by cloning the transmitting half of the channel in Listing 16-10:

Filename: src/main.rs

```
// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("from"),
        String::from("the"),
        String::from("thread"),
    ];
    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("more"),
        String::from("messages"),
        String::from("for"),
        String::from("you"),
    ];
    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("Got: {}", received);
}

// --snip--
```

Listing 16-11: Sending multiple messages from multiple producers

This time, before we create the first spawned thread, we call `clone` on the `s` channel. This will give us a new sending handle we can pass to the first spawned thread. This gives us two separate sending ends of the channel to a second spawned thread. This gives us two separate threads sending different messages to the receiving end of the channel.

When you run the code, your output should look something like this:

```
Got: hi
Got: more
Got: from
Got: messages
Got: for
Got: the
Got: thread
Got: you
```

You might see the values in another order; it depends on your system. This is interesting because it's concurrency interesting as well as difficult. If you experiment with `thread::spawn` values in the different threads, each run will be more nondeterministic and produce different results each time.

Now that we've looked at how channels work, let's look at a different method of communication between threads: `joinhandles`.

Shared-State Concurrency

Message passing is a fine way of handling concurrency, but it's not the only choice. Consider the slogan from the Go language documentation again: "communicate by sharing".

What would communicating by sharing memory look like? In addition, why would Go enthusiasts not use it and do the opposite instead?

In a way, channels in any programming language are similar to single ownership. When you transfer a value down a channel, you should no longer use that value. Shared ownership is like multiple ownership: multiple threads can access the same memory location simultaneously. As you saw in Chapter 15, where smart pointers made multiple ownership possible, shared ownership can add complexity because these different owners need management rules. These rules greatly assist in getting this management correct. For a mutex, one of the more common concurrency primitives for shared memory.

Using Mutexes to Allow Access to Data from One Thread at a Time

Mutex is an abbreviation for *mutual exclusion*, as in, a mutex allows only one thread to have exclusive access to data at any given time. To access the data in a mutex, a thread must first *lock* it by asking to acquire the mutex's *lock*. The lock is a data structure that is part of the mutex, and it keeps track of who currently has exclusive access to the data. Therefore, the mutex is said to be *guarding* the data it holds via the locking system.

Mutexes have a reputation for being difficult to use because you have to remember some rules:

- You must attempt to acquire the lock before using the data.
- When you're done with the data that the mutex guards, you must unlock it so that other threads can acquire the lock.

For a real-world metaphor for a mutex, imagine a panel discussion at a conference. Before a panelist can speak, they have to ask or signal that they want to speak into a microphone. When they get the microphone, they can talk for as long as they want, until they signal that they're finished and give the microphone to the next panelist who requests to speak. If a panelist forgets to give the microphone off when they're finished with it, no one else is able to speak. If a panelist gives the microphone to someone else while they're still speaking, the panel won't work as planned!

Management of mutexes can be incredibly tricky to get right, which is why so many people are enthusiastic about channels. However, thanks to Rust's type system and ownership rules, it's much easier to get locking and unlocking right.

The API of `Mutex<T>`

As an example of how to use a mutex, let's start by using a mutex in a single-threaded program, shown in Listing 16-12:

Filename: `src/main.rs`

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

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

Listing 16-12: Exploring the API of `Mutex<T>` in a single-threaded context for

As with many types, we create a `Mutex<T>` using the associated function `new`. Inside the mutex, we use the `lock` method to acquire the lock. This call will wait so it can't do any work until it's our turn to have the lock.

The call to `lock` would fail if another thread holding the lock panicked. In this case, we'd never be able to get the lock, so we've chosen to `unwrap` and have this thread block until it can get the lock.

After we've acquired the lock, we can treat the return value, named `num` in this example, as a reference to the data inside. The type system ensures that we acquire a lock in `m: Mutex<i32>` because `m` is not an `i32`, so we *must* acquire the lock to be able to use its value. We can't forget; the type system won't let us access the inner `i32` otherwise.

As you might suspect, `Mutex<T>` is a smart pointer. More accurately, the call to `lock` returns a smart pointer called `MutexGuard`. This smart pointer implements `Deref` to point at the data inside the mutex. The smart pointer also has a `Drop` implementation that releases the lock automatically when the smart pointer goes out of scope, which happens at the end of the inner scope. As a result, we don't risk forgetting to release the lock and blocking other threads because the lock release happens automatically.

After dropping the lock, we can print the mutex value and see that we were able to increment the `i32` to 6.

Sharing a `Mutex<T>` Between Multiple Threads

Now, let's try to share a value between multiple threads using `Mutex<T>`. We'll start by creating a new file, `src/main.rs`, and have them each increment a counter value by 1, so the counter goes from 0 to 5. These next few examples will have compiler errors, and we'll use those errors to learn how to use `Mutex<T>` and how Rust helps us use it correctly. Listing 16-13 has our starting code.

Filename: `src/main.rs`

```

use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}

```

Listing 16-13: Ten threads each increment a counter guarded by a `Mutex<T>`

We create a `counter` variable to hold an `i32` inside a `Mutex<T>`, as we did in Listing 16-12. We create 10 threads by iterating over a range of numbers. We use `thread::spawn` to create ten threads, each with its own copy of the closure. One thread moves the `counter` into the closure, acquires a lock on it, and adds 1 to the value. When the thread finishes running its closure, `num` will go out of scope and release the lock so that other threads can acquire it.

In the main thread, we collect all the join handles. Then, as we did in Listing 16-12, we wait for each handle to make sure all the threads finish. At that point, the main thread prints the result of this program.

We hinted that this example wouldn't compile. Now let's find out why!

```

error[E0382]: capture of moved value: `counter`
--> src/main.rs:10:27
|
9 |         let handle = thread::spawn(move || {
|                         ----- value moved (into closure at line 10)
10|             let mut num = counter.lock().unwrap();
|                         ^^^^^^^^ value captured here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex`
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:21:29
|
9 |         let handle = thread::spawn(move || {
|                         ----- value moved (into closure at line 21)
...
21|     println!("Result: {}", *counter.lock().unwrap());
|                         ^^^^^^^^ value used here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex`
which does not implement the `Copy` trait

error: aborting due to 2 previous errors

```

The error message states that the `counter` value is moved into the closure; it cannot be copied back out.

we call `lock`. That description sounds like what we wanted, but it's not allowed.

Let's figure this out by simplifying the program. Instead of making 10 threads, let's make two threads without a loop and see what happens. Replace the first `f` with this code instead:

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();

        *num += 1;
    });
    handles.push(handle);

    let handle2 = thread::spawn(move || {
        let mut num2 = counter.lock().unwrap();

        *num2 += 1;
    });
    handles.push(handle2);

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

We make two threads and change the variable names used with the second `num2`. When we run the code this time, compiling gives us the following:

```
error[E0382]: capture of moved value: `counter`
--> src/main.rs:16:24
|
8 |     let handle = thread::spawn(move || {
|                         ----- value moved (into closure)
...
16 |         let mut num2 = counter.lock().unwrap();
|                         ^^^^^^^^ value captured here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex`
which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`
--> src/main.rs:26:29
|
8 |     let handle = thread::spawn(move || {
|                         ----- value moved (into closure)
...
26 |     println!("Result: {}", *counter.lock().unwrap());
|                         ^^^^^^^^ value used here after move
|
= note: move occurs because `counter` has type `std::sync::Mutex`
which does not implement the `Copy` trait

error: aborting due to 2 previous errors
```

Aha! The first error message indicates that `counter` is moved into the closure associated with `handle`. That move is preventing us from capturing `counter`.

lock on it and store the result in `num2` in the second thread! So Rust is telling ownership of `counter` into multiple threads. This was hard to see earlier because in a loop, and Rust can't point to different threads in different iterations of the compiler error with a multiple-ownership method we discussed in Chapter 1

Multiple Ownership with Multiple Threads

In Chapter 15, we gave a value multiple owners by using the smart pointer `Rc` for reference counted value. Let's do the same here and see what happens. We'll use `Rc<T>` in Listing 16-14 and clone the `Rc<T>` before moving ownership to the threads. Once we've seen the errors, we'll also switch back to using the `for` loop, and we'll keep the closure.

Filename: src/main.rs

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 16-14: Attempting to use `Rc<T>` to allow multiple threads to own the same value

Once again, we compile and get... different errors! The compiler is teaching us about multiple ownership.

```
error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>: std::marker::Send` is not satisfied in `[closure@src/main.rs:11:36 .. 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`
--> src/main.rs:11:22
 |
11 |         let handle = thread::spawn(move || {
|             ^^^^^^^^^^^^^^ `std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads safely
|             |
|             = help: within `[closure@src/main.rs:11:36 .. 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`, the trait `std::marker::Send` is not implemented for `std::rc::Rc<std::sync::Mutex<i32>>`
|             = note: required because it appears within the type
|             `[closure@src/main.rs:11:36 .. 15:10 counter:std::rc::Rc<std::sync::Mutex<i32>>]`
|             = note: required by `std::thread::spawn`
```

Wow, that error message is very wordy! Here are some important parts to focus on. The error says ``std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads`. The reason for this is in the next important part to focus on, the error message.

message says the trait bound `Send` is not satisfied. We'll talk about section: it's one of the traits that ensures the types we use with threads are safe in concurrent situations.

Unfortunately, `Rc<T>` is not safe to share across threads. When `Rc<T>` manages a count, it adds to the count for each call to `clone` and subtracts from the count when it's dropped. But it doesn't use any concurrency primitives to make sure that changes can't be interrupted by another thread. This could lead to wrong counts—subtle bugs that lead to memory leaks or a value being dropped before we're done with it. `Arc<T>` is similar to `Rc<T>` but one that makes changes to the reference count in a thread-safe way.

Atomic Reference Counting with `Arc<T>`

Fortunately, `Arc<T>` is a type like `Rc<T>` that is safe to use in concurrent situations. It's called *atomic*, meaning it's an *atomically reference counted* type. Atomics are an add primitive that we won't cover in detail here: see the standard library `std::sync::atomic` for more details. At this point, you just need to know that atomic types are safe to share across threads.

You might then wonder why all primitive types aren't atomic and why standard types are implemented to use `Arc<T>` by default. The reason is that thread safety comes at a performance penalty that you only want to pay when you really need to. If you're just performing operations on values within a single thread, your code can run faster if it doesn't have to rely on atomic primitives.

Let's return to our example: `Arc<T>` and `Rc<T>` have the same API, so we'll start by changing the `use` line, the call to `new`, and the call to `clone`. The code in Listing 16-15 will compile and run:

Filename: `src/main.rs`

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 16-15: Using an `Arc<T>` to wrap the `Mutex<T>` to be able to share over multiple threads

This code will print the following:

```
Result: 10
```

We did it! We counted from 0 to 10, which may not seem very impressive, but about `Mutex<T>` and thread safety. You could also use this program's struct for more complicated operations than just incrementing a counter. Using this strategy, you can calculate into independent parts, split those parts across threads, and then have each thread update the final result with its part.

Similarities Between `RefCell<T> / Rc<T>` and `Mutex<T> / Arc<T>`

You might have noticed that `counter` is immutable but we could get a mutable value inside it; this means `Mutex<T>` provides interior mutability, as the `cell` does. In fact, we can do the same thing with `Rc<T>` in Chapter 15 to allow us to mutate content using `Mutex<T>` to mutate contents inside an `Arc<T>`.

Another detail to note is that Rust can't protect you from all kinds of logic errors when using `Mutex<T>`. Recall in Chapter 15 that using `Rc<T>` came with the risk of creating `phantom references`, where two `Rc<T>` values refer to each other, causing memory leaks. Similarly, using `Mutex<T>` comes with the risk of creating *deadlocks*. These occur when an operation needs to lock both mutexes, but both threads have each acquired one of the locks, causing them to wait for each other. If you're interested in deadlocks, try creating a Rust program that has a deadlock; there are many mitigation strategies for mutexes in any language and have a go at implementing them. The standard library API documentation for `Mutex<T>` and `MutexGuard` offers useful information.

We'll round out this chapter by talking about the `Send` and `Sync` traits and how they work with custom types.

Extensible Concurrency with the `Sync` and `Send` Traits

Interestingly, the Rust language has *very few* concurrency features. Almost every feature we've talked about so far in this chapter has been part of the standard library or part of the language itself. Your options for handling concurrency are not limited to the language or the standard library; you can write your own concurrency features or use those written by others.

However, two concurrency concepts are embedded in the language: the `Sync` and `Send` traits.

Allowing Transference of Ownership Between Threads with `Send`

The `Send` marker trait indicates that ownership of the type implementing `Send` can be transferred between threads. Almost every Rust type is `Send`, but there are some exceptions. For example, `Rc<T>` is not `Send` because if you cloned an `Rc<T>` value and tried to transfer it to another thread, both threads might update the reference count at the same time, causing a race condition. `Rc<T>` is implemented for use in single-threaded situations where you don't care about thread-safe performance penalty.

Therefore, Rust's type system and trait bounds ensure that you can never accidentally move a `Rc<T>` value across threads unsafely. When we tried to do this in Listing 16-1, we got a compile error because the trait `Send` is not implemented for `Rc<Mutex<i32>>`. When we switched to `Arc`, the code compiled.

Any type composed entirely of `Send` types is automatically marked as `Send`. Primitive types are `Send`, aside from raw pointers, which we'll discuss in Chapter 17.

Allowing Access from Multiple Threads with `Sync`

The `Sync` marker trait indicates that it is safe for the type implementing `Sync` to be sent between threads. In other words, any type `T` is `Sync` if `&T` (a reference to `T`) is `Sync`, meaning the reference can be sent safely to another thread. Similar to `Send`, `Sync` is a marker trait, meaning types composed entirely of types that are `Sync` are also `Sync`.

The smart pointer `Rc<T>` is also not `Sync` for the same reasons that it's not `Send` (which we talked about in Chapter 15) and the family of related `Cell<T>` types is not `Sync`. The implementation of borrow checking that `RefCell<T>` does at runtime is not `Sync`, so you can't share access to a `RefCell` across threads. The smart pointer `Mutex<T>` is `Sync` and can be used to share access with multiple threads. See the "Sharing a `Mutex<T>` Between Multiple Threads" section.

Implementing `Send` and `Sync` Manually Is Unsafe

Because types that are made up of `Send` and `Sync` traits are automatically `Send` and `Sync`, you don't have to implement those traits manually. As marker traits, they don't even have to implement methods. They're just useful for enforcing invariants related to concurrency.

Manually implementing these traits involves implementing unsafe Rust code. If you've never written unsafe Rust code in Chapter 19; for now, the important information is that building safe concurrent programs that work well in real-world situations requires careful thought to uphold safety guarantees. [The Rustonomicon](#) has more information about these guarantees and how to implement them.

Summary

This isn't the last you'll see of concurrency in this book: the project in Chapter 19 is a good example of how to apply the concepts in this chapter in a more realistic situation than the smaller examples in this chapter.

As mentioned earlier, because very little of how Rust handles concurrency is implemented in the standard library, many concurrency solutions are implemented as crates. These evolve more rapidly than the standard library, so be sure to search online for the current, state-of-the-art solutions for handling concurrency in multithreaded situations.

The Rust standard library provides channels for message passing and smart pointers like `Mutex<T>` and `Arc<T>`, that are safe to use in concurrent contexts. The type checker ensures that the code using these solutions won't end up with data races or undefined behavior. Once you get your code to compile, you can rest assured that it will work correctly in concurrent environments. Concurrent programming is no longer a concept to be afraid of: go forth and program concurrently, fearlessly!

Next, we'll talk about idiomatic ways to model problems and structure solutions as programs get bigger. In addition, we'll discuss how Rust's idioms relate to the concepts you may be familiar with from object-oriented programming.

Object Oriented Programming Features in Rust

Object-oriented programming (OOP) is a way of modeling programs. Objects were first developed in the 1960s. Those objects influenced Alan Kay's programming architecture in the 1970s, which became the basis for Smalltalk, the first true object-oriented language.

messages to each other. He coined the term *object-oriented programming* in architecture. Many competing definitions describe what OOP is; some define it as object oriented, but other definitions would not. In this chapter, we'll explore characteristics that are commonly considered object oriented and how those translate to idiomatic Rust. We'll then show you how to implement an object in Rust and discuss the trade-offs of doing so versus implementing a solution's strengths instead.

Characteristics of Object-Oriented Languages

There is no consensus in the programming community about what features should be considered object oriented. Rust is influenced by many programming paradigms; for example, we explored the features that came from functional programming. Arguably, OOP languages share certain common characteristics, namely objects, inheritance, and encapsulation. Let's look at what each of those characteristics means and why they're important.

Objects Contain Data and Behavior

The book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994) describes objects as "the basic building blocks of OOP". The *Gang of Four* book, is a catalog of object-oriented design patterns. It defines an object as:

Object-oriented programs are made up of objects. An *object* packages both data and procedures that operate on that data. The procedures are typically called *operations*.

Using this definition, Rust is object oriented: structs and enums have data, and methods on structs and enums provide behavior. Even though structs and enums with methods provide the same functionality, according to the Gang of Four's definition, they are objects.

Encapsulation that Hides Implementation Details

Another aspect commonly associated with OOP is the idea of *encapsulation*, which means that implementation details of an object aren't accessible to code using that object. The way to interact with an object is through its public API; code using the object can't reach into the object's internals and change data or behavior directly. This enables us to change and refactor an object's internals without needing to change the code that uses it.

We discussed how to control encapsulation in Chapter 7: we can use the `pub` keyword to indicate which modules, types, functions, and methods in our code should be public, and everything else is private. For example, we can define a struct `AveragedCollection` containing a vector of `i32` values. The struct can also have a field that contains the average of the values in the vector, meaning the average doesn't have to be computed on every iteration. In other words, `AveragedCollection` will cache the calculated average. Listing 17-1 has the definition of the `AveragedCollection` struct:

Filename: `src/lib.rs`

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Listing 17-1: An `AveragedCollection` struct that maintains a list of integers in the collection

The struct is marked `pub` so that other code can use it, but the fields within are private. This is important in this case because we want to ensure that whenever an item is removed from the list, the average is also updated. We do this by implementing `average` methods on the struct, as shown in Listing 17-2:

Filename: `src/lib.rs`

```
impl AveragedCollection {
    pub fn add(&mut self, value: i32) {
        self.list.push(value);
        self.update_average();
    }

    pub fn remove(&mut self) -> Option<i32> {
        let result = self.list.pop();
        match result {
            Some(value) => {
                self.update_average();
                Some(value)
            },
            None => None,
        }
    }

    pub fn average(&self) -> f64 {
        self.average
    }

    fn update_average(&mut self) {
        let total: i32 = self.list.iter().sum();
        self.average = total as f64 / self.list.len() as f64;
    }
}
```

Listing 17-2: Implementations of the public methods `add`, `remove`, and `average` for `AveragedCollection`

The public methods `add`, `remove`, and `average` are the only ways to modify `AveragedCollection`. When an item is added to `list` using the `add` method or removed using the `remove` method, the implementations of each call the private `update_average` method to update the `average` field as well.

We leave the `list` and `average` fields private so there is no way for external code to change them directly; otherwise, the `average` field might become out of sync with the `list` changes. The `average` method returns the value in the `average` field without modifying it.

Because we've encapsulated the implementation details of the struct `AveragedCollection`, we can easily change aspects, such as the data structure, in the future. For instance, we could use `HashSet` instead of a `Vec` for the `list` field. As long as the signatures of the `add` and `remove` public methods stay the same, code using `AveragedCollection` will still work. If we made `list` public instead, this wouldn't necessarily be the case: `HashMap` uses

different methods for adding and removing items, so the external code would if it were modifying `list` directly.

If encapsulation is a required aspect for a language to be considered object-oriented, then the language must meet that requirement. The option to use `pub` or not for different parts of a struct is one way to provide encapsulation of implementation details.

Inheritance as a Type System and as Code Sharing

Inheritance is a mechanism whereby an object can inherit from another object, gaining the parent object's data and behavior without you having to define them again.

If a language must have inheritance to be an object-oriented language, then there is no way to define a struct that inherits the parent struct's fields and methods. However, if you're used to having inheritance in your programming toolbox, there are other ways to achieve similar results in Rust, depending on your reason for reaching for inheritance in the first place.

You choose inheritance for two main reasons. One is for reuse of code: you want to reuse some particular behavior for one type, and inheritance enables you to reuse that implementation for another type. You can share Rust code using default trait method implementations, for example, as you saw in Listing 10-14 when we added a default implementation of the `Summary` trait to the `String` type. Any type implementing the `Summary` trait would have the `summary()` method available on it without any further code. This is similar to a parent class having a method and an inheriting child class also having the implementation of that method. You can also override the default implementation of the `summarize` method when we implement the `Summary` trait for our own type, which is similar to a child class overriding the implementation of a method from its parent class.

The other reason to use inheritance relates to the type system: to enable a class to have multiple types at the same places as the parent type. This is also called *polymorphism*, which refers to the ability to substitute multiple objects for each other at runtime if they share certain characteristics.

Polymorphism

To many people, polymorphism is synonymous with inheritance. But it's a more general concept that refers to code that can work with data of multiple types. In fact, those types are generally subclasses.

Rust instead uses generics to abstract over different possible types and traits, placing constraints on what those types must provide. This is sometimes called *behavioral polymorphism*.

Inheritance has recently fallen out of favor as a programming design solution in many languages because it's often at risk of sharing more code than necessary. Subclasses always share all characteristics of their parent class but will do so with inheritance, making the program's design less flexible. It also introduces the possibility of calling methods that don't make sense or that cause errors because the methods don't apply to the subclass. Some languages will only allow a subclass to inherit from one class, further limiting the expressiveness of a program's design.

For these reasons, Rust takes a different approach, using trait objects instead of inheritance. Let's look at how trait objects enable polymorphism in Rust.

Using Trait Objects that Allow for Values of Diff

In Chapter 8, we mentioned that one limitation of vectors is that they can store only one type. We created a workaround in Listing 8-10 where we defined a `Spreadsheet` struct with variants to hold integers, floats, and text. This meant we could store different types in the same vector cell and still have a vector that represented a row of cells. This is a perfectly reasonable solution, but it means that interchangeable items are a fixed set of types that we know when our code is written.

However, sometimes we want our library user to be able to extend the set of interchangeable items to fit a particular situation. To show how we might achieve this, we'll create an example of a GUI tool that iterates through a list of items, calling a `draw` method on each item to draw it to the screen—a common technique for GUI tools. We'll create a library crate named `gui` that contains the structure of a GUI library. This crate might include some types for common components, such as `Button` or `TextField`. In addition, `gui` users will want to create their own components that can be drawn: for instance, one programmer might add an `Image` and another might add a `Table`.

We won't implement a fully fledged GUI library for this example but will show how to structure it together. At the time of writing the library, we can't know and define all the different types that programmers might want to create. But we do know that `gui` needs to keep track of different types, and it needs to call a `draw` method on each of these different types. The programmer who uses `gui` doesn't need to know exactly what will happen when we call the `draw` method; they just need to know that `gui` will have that method available for us to call.

To do this in a language with inheritance, we might define a class named `Component` and have other classes inherit from it. Then we could define a `draw` method named `draw` on it. The other classes, such as `Button`, `Image`, and `Table`, would inherit from `Component` and thus inherit the `draw` method. They could each override the `draw` method to define their custom behavior, but the framework could treat all of them as `Component` instances and call `draw` on them. But because Rust doesn't have inheritance, we need another way to structure the `gui` library to allow users to extend it without changing the core code.

Defining a Trait for Common Behavior

To implement the behavior we want `gui` to have, we'll define a trait named `Draw` that defines a `draw` method named `draw`. Then we can define a vector that takes a *trait object*. A trait object is a pointer to an implementation of a trait, similar to a pointer to a function or a pointer to a variable. It's a sort of pointer, such as a `&` reference or a `Box<T>` smart pointer, and then we can add a `dyn` keyword to indicate that it's a trait object. (We'll talk about the reason trait objects must be `dyn` in the section "Dynamically Sized Types & Sized".) We can use trait objects to abstract away the concrete type. Wherever we use a trait object, Rust's type system will ensure that any value used in that context will implement the trait object's trait. Consequently, we can use trait objects to know all the possible types at compile time.

We've mentioned that in Rust, we refrain from calling structs and enums "objects" because they're not objects in the sense that they don't have behavior. In a struct or enum, the data in the structure and the behavior in the methods are separate. In contrast, traits and the behavior in `impl` blocks are combined into one concept. Whereas in other languages, the data and behavior of an object are combined into one concept called an object. However, trait objects are objects in the sense that they combine data and behavior. But trait objects are not traditional objects in that we can't add data to a trait object. Trait objects are more like objects in other languages: their specific purpose is to allow abstraction across multiple types.

Listing 17-3 shows how to define a trait named `Draw` with one method named `draw`.

Filename: `src/lib.rs`

```
pub trait Draw {
    fn draw(&self);
}
```

Listing 17-3: Definition of the `Draw` trait

This syntax should look familiar from our discussions on how to define traits. Listing 17-3 defines a trait named `Draw` with one method, `draw`. Listing 17-4 comes some new syntax: Listing 17-4 defines a struct named `Screen` that holds a vector of components. This vector is of type `Box<dyn Draw>`, which is a trait object; it's inside a `Box` that implements the `Draw` trait.

Filename: `src/lib.rs`

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

Listing 17-4: Definition of the `Screen` struct with a `components` field holding components that implement the `Draw` trait

On the `Screen` struct, we'll define a method named `run` that will call the `draw` method on each of its `components`, as shown in Listing 17-5:

Filename: `src/lib.rs`

```
impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-5: A `run` method on `Screen` that calls the `draw` method on each component

This works differently than defining a struct that uses a generic type parameter. A generic type parameter can only be substituted with one concrete type at a time. Trait objects allow for multiple concrete types to fill in for the trait object at runtime. Listing 17-6 shows how we could have defined the `Screen` struct using a generic type and a trait bound.

Filename: `src/lib.rs`

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
}

impl<T> Screen<T>
where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

Listing 17-6: An alternate implementation of the `Screen` struct and its `run` method using trait bounds

This restricts us to a `Screen` instance that has a list of components all of type `TextField`. If you'll only ever have homogeneous collections, using generics is preferable because the definitions will be monomorphized at compile time to types.

On the other hand, with the method using trait objects, one `Screen` instance contains a `Box<Button>` as well as a `Box<TextField>`. Let's look at how this talk about the runtime performance implications.

Implementing the Trait

Now we'll add some types that implement the `Draw` trait. We'll provide the implementation for `Button` here, but actually implementing a GUI library is beyond the scope of this book, so the `draw` method is just a placeholder. To imagine what the implementation might look like, consider that the `Button` struct might have fields for `width`, `height`, and `label`, as shown in Listing 17-7.

Filename: `src/lib.rs`

```
pub struct Button {
    pub width: u32,
    pub height: u32,
    pub label: String,
}

impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

Listing 17-7: A `Button` struct that implements the `Draw` trait

The `width`, `height`, and `label` fields on `Button` will differ from the fields on other types, such as a `TextField` type, that might have those fields plus a `placeholder` field. Other types we want to draw on the screen will implement the `Draw` trait but will likely have their own `draw` methods that call the `draw` method on `Button`. (This is similar to how `draw` is a trait on `Image` but `Image` has its own `draw` method to define how to draw that particular type, as `Image` has its own specific GUI code, which is beyond the scope of this chapter). The `Button` type, for its part, only needs an `impl` block containing methods related to what happens when a `draw` method is called on it. These kinds of methods won't apply to types like `TextField`.

If someone using our library decides to implement a `SelectBox` struct that has a list of `options` fields, they implement the `Draw` trait on the `SelectBox` type as we saw in Listing 17-8:

Filename: `src/main.rs`

```

extern crate gui;
use gui::Draw;

struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}

impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}

```

Listing 17-8: Another crate using `gui` and implementing the `Draw` trait on a

Our library's user can now write their `main` function to create a `Screen` instance, they can add a `SelectBox` and a `Button` by putting each in a `Box<` object. They can then call the `run` method on the `screen` instance, which will run the components. Listing 17-9 shows this implementation:

Filename: `src/main.rs`

```

use gui::{Screen, Button};

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No")
                ],
            }),
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}

```

Listing 17-9: Using trait objects to store values of different types that implement the `Draw` trait.

When we wrote the library, we didn't know that someone might add the `SelectBox` type. The `Screen` implementation was able to operate on the new type and draw it because the `SelectBox` type implements the `Draw` type, which means it implements the `draw` method.

This concept—of being concerned only with the messages a value responds to without caring about its concrete type—is similar to the concept *duck typing* in dynamically typed languages like Python. If an animal quacks like a duck and walks like a duck, then it must be a duck! In the implementation of `Screen` in Listing 17-5, `run` doesn't need to know what the concrete type of each component is; it just needs to know that each component implements the `Draw` trait. By specifying `Box<dyn Draw>` as the type of the `components` vector, we've defined `Screen` to need values that we can call the `draw` method on.

The advantage of using trait objects and Rust's type system to write code similar to Listing 17-9 is that we never have to check whether a value implements a particular trait. We don't worry about getting errors if a value doesn't implement a method but we can still compile our code if the values don't implement the traits that the trait object requires.

For example, Listing 17-10 shows what happens if we try to create a `Screen` component:

Filename: `src/main.rs`

```
extern crate gui;
use gui::Screen;

fn main() {
    let screen = Screen {
        components: vec![
            Box::new(String::from("Hi")),
        ],
    };
    screen.run();
}
```

Listing 17-10: Attempting to use a type that doesn't implement the trait object

We'll get this error because `String` doesn't implement the `Draw` trait:

```
error[E0277]: the trait bound `std::string::String: gui::Draw` is not satisfied
--> src/main.rs:7:13
  |
7 |         Box::new(String::from("Hi")),
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `gui::Draw` is not implemented for `std::string::String`
  |
  = note: required for the cast to the object type `gui::Draw`
```

This error lets us know that either we're passing something to `Screen` we didn't expect or we should pass a different type or we should implement `Draw` on `String` so we can call `draw` on it.

Trait Objects Perform Dynamic Dispatch

Recall in the "Performance of Code Using Generics" section in Chapter 10 our discussion of monomorphization process performed by the compiler when we use trait bounds. When we use trait objects, the compiler generates nongeneric implementations of functions and methods that we use in place of a generic type parameter. The code that results from doing *static dispatch*, which is when the compiler knows what method you're calling. This is opposed to *dynamic dispatch*, which is when the compiler can't tell at runtime which method you're calling. In dynamic dispatch cases, the compiler emits code that dispatches to figure out which method to call.

When we use trait objects, Rust must use dynamic dispatch. The compiler does this by emitting code that might be used with the code that is using trait objects, so it doesn't know which method is implemented on which type to call. Instead, at runtime, Rust uses the pointer to know which method to call. There is a runtime cost when this lookup happens compared with static dispatch. Dynamic dispatch also prevents the compiler from choosing better code, which in turn prevents some optimizations. However, we did get extra functionality that we wrote in Listing 17-5 and were able to support in Listing 17-9, so it's worth the cost.

Object Safety Is Required for Trait Objects

You can only make *object-safe* traits into trait objects. Some complex rules go into what makes a trait object safe, but in practice, only two rules are relevant. A trait object is safe if all the methods defined in the trait have the following properties:

- The return type isn't `Self`.
- There are no generic type parameters.

The `Self` keyword is an alias for the type we're implementing the traits or methods for. It must be object safe because once you've used a trait object, Rust no longer knows what type it's implementing that trait. If a trait method returns the concrete `Self` type, the compiler forgets the exact type that `Self` is, there is no way the method can use the type information. The same is true of generic type parameters that are filled in with concrete types when the trait is used: the concrete types become part of the type that implements the trait. If the concrete type is forgotten through the use of a trait object, there is no way to know what generic type parameters were used with it.

An example of a trait whose methods are not object safe is the standard library's signature for the `clone` method in the `Clone` trait looks like this:

```
pub trait Clone {  
    fn clone(&self) -> Self;  
}
```

The `String` type implements the `Clone` trait, and when we call the `clone` method of `String` we get back an instance of `String`. Similarly, if we call `clone` on `Vec`, we get back an instance of `Vec`. The signature of `clone` needs to know what type it's cloning, because that's the return type.

The compiler will indicate when you're trying to do something that violates the rules in regard to trait objects. For example, let's say we tried to implement the `Screen` trait from Listing 17-4 to hold types that implement the `Clone` trait instead of the `Draw` trait,

```
pub struct Screen {  
    pub components: Vec<Box<dyn Clone>>,  
}
```

We would get this error:

```
error[E0038]: the trait `std::clone::Clone` cannot be made into an object  
--> src/lib.rs:2:5  
|  
2 |     pub components: Vec<Box<dyn Clone>>,  
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait `std::clone::Clone`  
made into an object  
|  
= note: the trait cannot require that `Self : Sized`
```

This error means you can't use this trait as a trait object in this way. If you're interested in learning more about object safety, see [Rust RFC 255](#).

Implementing an Object-Oriented Design Pattern

The *state pattern* is an object-oriented design pattern. The crux of the pattern is that some internal state, which is represented by a set of *state objects*, and the value of

based on the internal state. The state objects share functionality: in Rust, of and traits rather than objects and inheritance. Each state object is responsible for governing when it should change into another state. The value that it knows nothing about the different behavior of the states or when to transition.

Using the state pattern means when the business requirements of the program need to change the code of the value holding the state or the code that uses state objects. Let's look at an example of the state design pattern and how to implement it.

We'll implement a blog post workflow in an incremental way. The blog's final state will be:

1. A blog post starts as an empty draft.
2. When the draft is done, a review of the post is requested.
3. When the post is approved, it gets published.
4. Only published blog posts return content to print, so unapproved posts remain empty.

Any other changes attempted on a post should have no effect. For example, if we try to add text to a draft blog post before we've requested a review, the post should remain an empty string.

Listing 17-11 shows this workflow in code form: this is an example usage of the state pattern in a library crate named `blog`. This won't compile yet because we haven't implemented the `Post` type in the `blog` crate yet.

Filename: `src/main.rs`

```
extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

    post.approve();
    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Listing 17-11: Code that demonstrates the desired behavior we want our blog to exhibit.

We want to allow the user to create a new draft blog post with `Post::new()`. Then, we want to be able to add text to the blog post while it's in the draft state. If we try to get the content of the post immediately, before approval, nothing should happen because the post is still a draft. We can use `assert_eq!` in the code for demonstration purposes. An excellent unit test is to assert that a draft blog post returns an empty string from the `content` method. It's up to you to write tests for this example.

Next, we want to enable a request for a review of the post, and we want `content` to return an empty string while waiting for the review. When the post receives approval, its state changes to published, meaning the text of the post will be returned when `content` is called.

Notice that the only type we're interacting with from the crate is the `Post` type. The `Post` type represents a blog post and will hold a value that will be one of three state objects represented by the `Draft`, `WaitingForReview`, and `Published` variants. The `Post` type represents a post can be in—draft, waiting for review, or published. Changing from one state to another will be managed internally within the `Post` type. The states change in response to external events like `add_text`, `request_review`, and `approve`.

called by our library's users on the `Post` instance, but they don't have to make it directly. Also, users can't make a mistake with the states, like publishing a post while it's still a draft.

Defining `Post` and Creating a New Instance in the Draft State

Let's get started on the implementation of the library! We know we need a `Post` struct that holds some content, so we'll start with the definition of the struct and an associated function to create an instance of `Post`, as shown in Listing 17-12. We'll also implement the `State` trait. Then `Post` will hold a trait object of `Box<dyn State>` inside an `Option` named `state`. You'll see why the `Option<T>` is necessary in a bit.

Filename: `src/lib.rs`

```
pub struct Post {
    state: Option<Box<dyn State>>,
    content: String,
}

impl Post {
    pub fn new() -> Post {
        Post {
            state: Some(Box::new(Draft {})),
            content: String::new(),
        }
    }
}

trait State {}

struct Draft {}

impl State for Draft {}
```

Listing 17-12: Definition of a `Post` struct and a `new` function that creates a `Post` with a `State` trait, and a `Draft` struct

The `State` trait defines the behavior shared by different post states, and thus `PendingReview`, and `Published` states will all implement the `State` trait. For now, we'll just define the `State` trait with no methods, and we'll start by defining just the `Draft` state because that's the easiest state for a post to start in.

When we create a new `Post`, we set its `state` field to a `Some` value that holds a trait object of `Draft`. This ensures whenever we create a new `Post`, it will start out as a draft. Because the `state` field of `Post` is private, we can't publish a `Post` in any other state! In the `Post::new` function, we set the `content` field to an empty `String`.

Storing the Text of the Post Content

Listing 17-11 showed that we want to be able to call a method named `add_text` that is then added to the text content of the blog post. We implement this as an associated function on the `Post` type, exposing the `content` field as `pub`. This means we can implement a method that changes how the `content` field's data is read. The `add_text` method is pretty straightforward, so we'll leave the implementation in Listing 17-13 to the `impl Post` block:

Filename: `src/lib.rs`

```
impl Post {  
    // --snip--  
    pub fn add_text(&mut self, text: &str) {  
        self.content.push_str(text);  
    }  
}
```

Listing 17-13: Implementing the `add_text` method to add text to a post's `content` field.

The `add_text` method takes a mutable reference to `self`, because we're changing the instance that we're calling `add_text` on. We then call `push_str` on the `String` in the `text` argument to add to the saved `content`. This behavior doesn't depend on the state of the post, so it's not part of the state pattern. The `add_text` method doesn't change the `state` field at all, but it is part of the behavior we want to support.

Ensuring the Content of a Draft Post Is Empty

Even after we've called `add_text` and added some content to our post, we still have the `content` method to return an empty string slice because the post is still in the draft state, as shown in Listing 17-11. For now, let's implement the `content` method with the simplest possible implementation that satisfies this requirement: always returning an empty string slice. We'll change this later when we add the ability to change a post's state so it can be published. So far, posts can only be published if they have content, so the post content should always be empty. Listing 17-14 shows this placeholder implementation.

Filename: `src/lib.rs`

```
impl Post {  
    // --snip--  
    pub fn content(&self) -> &str {  
        ""  
    }  
}
```

Listing 17-14: Adding a placeholder implementation for the `content` method that always returns an empty string slice

With this added `content` method, everything in Listing 17-11 up to line 8 works correctly.

Requesting a Review of the Post Changes Its State

Next, we need to add functionality to request a review of a post, which should change the `state` from `Draft` to `PendingReview`. Listing 17-15 shows this code:

Filename: `src/lib.rs`

```

impl Post {
    // --snip--
    pub fn request_review(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.request_review())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReview {})
    }
}

struct PendingReview {}

impl State for PendingReview {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

```

Listing 17-15: Implementing `request_review` methods on `Post` and the `State` trait

We give `Post` a public method named `request_review` that will take a mutable reference to `self`. Then we call an internal `request_review` method on the current state of `Post`. This internal `request_review` method consumes the current state and returns a new state.

We've added the `request_review` method to the `State` trait; all types that implement the `State` trait now need to implement the `request_review` method. Note that rather than taking a mutable reference to `self` as the first parameter of the method, we have `self: Box<Self>` as the parameter. This means that the method is only valid when called on a `Box` holding the type. This syntax is called a `boxed trait bound`, invalidating the old state so the state value of the `Post` can truly be consumed.

To consume the old state, the `request_review` method needs to take ownership of it. This is where the `Option` in the `state` field of `Post` comes in: we call the `take` method on the `state` field to remove the `Some` value out of the `state` field and leave a `None` in its place, because `None` is a valid value for unpopulated fields in structs. This lets us move the `state` value out of `Post` and into `PendingReview`. Then we'll set the post's `state` value to the result of this operation.

We need to set `state` to `None` temporarily rather than setting it directly with `self.state = self.state.request_review();` to get ownership of the state. If we did this, `Post` would own the old `state` value after we've transformed it into a new `PendingReview` state.

The `request_review` method on `Draft` needs to return a new, boxed instance of the `PendingReview` struct, which represents the state when a post is waiting for review. The `PendingReview` struct also implements the `request_review` method but does something different: instead of returning a new state, it returns itself, because when we request a review of a post in the `PendingReview` state, it should stay in the `PendingReview` state.

Now we can start seeing the advantages of the state pattern: the `request_review` method always does the same thing no matter what its `state` value is. Each state is responsible for its own implementation of the `request_review` method.

We'll leave the `content` method on `Post` as is, returning an empty string since we don't care about the content of a post while it's pending review.

Post in the PendingReview state as well as in the Draft state, but we want the PendingReview state. Listing 17-11 now works up to line 11!

Adding the approve Method that Changes the Behavior of cc

The approve method will be similar to the request_review method: it will say that the current state says it should have when that state is approved, as shown:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn approve(&mut self) {
        if let Some(s) = self.state.take() {
            self.state = Some(s.approve())
        }
    }
}

trait State {
    fn request_review(self: Box<Self>) -> Box<dyn State>;
    fn approve(self: Box<Self>) -> Box<dyn State>;
}

struct Draft {}

impl State for Draft {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}

struct PendingReview {}

impl State for PendingReview {
    // --snip--
    fn approve(self: Box<Self>) -> Box<dyn State> {
        Box::new(Published {})
    }
}

struct Published {}

impl State for Published {
    fn request_review(self: Box<Self>) -> Box<dyn State> {
        self
    }

    fn approve(self: Box<Self>) -> Box<dyn State> {
        self
    }
}
```

Listing 17-16: Implementing the approve method on Post and the State trait

We add the approve method to the State trait and add a new struct that implements the Published state.

Similar to request_review, if we call the approve method on a Draft, it will return self. When we call approve on PendingReview, it returns a new instance of the Published struct. The Published struct implements the State trait, and

request_review method and the approve method, it returns itself, because the Published state in those cases.

Now we need to update the content method on Post: if the state is Published, return the value in the post's content field; otherwise, we want to return an empty string. Listing 17-17:

Filename: src/lib.rs

```
impl Post {
    // --snip--
    pub fn content(&self) -> &str {
        self.state.as_ref().unwrap().content(&self)
    }
    // --snip--
}
```

Listing 17-17: Updating the content method on Post to delegate to a content method on State

Because the goal is to keep all these rules inside the structs that implement the content method on the value in state and pass the post instance (that is, the post itself) to the content method on state. Then we return the value that is returned from using the content method on state.

We call the as_ref method on the Option because we want a reference to the Option rather than ownership of the value. Because state is an Option<&Box<dyn State>>, we call as_ref, an Option<&Box<dyn State>>, is returned. If we didn't call as_ref, we'd get a borrow error because we can't move state out of the borrowed &self of the function.

We then call the unwrap method, which we know will never panic, because we've ensured that Post ensure that state will always contain a Some value when those methods are called. This is one of the cases we talked about in the “Cases When You Have More Information Than the Compiler” section of Chapter 9 when we know that a None value is never possible. The compiler isn't able to understand that.

At this point, when we call content on the &Box<dyn State>, deref coercion will turn the Box into a reference to the State trait. That means we need to add content to the State trait definition. Listing 17-18:

Filename: src/lib.rs

```
trait State {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        ""
    }
}

// --snip--
struct Published {}

impl State for Published {
    // --snip--
    fn content<'a>(&self, post: &'a Post) -> &'a str {
        &post.content
    }
}
```

Listing 17-18: Adding the content method to the state trait

We add a default implementation for the `content` method that returns an `&str`. This means we don't need to implement `content` on the `Draft` and `PendingReview` structs. The `Published` struct will override the `content` method and return the value in its `content` field.

Note that we need lifetime annotations on this method, as we discussed in Chapter 17. A reference to a `post` as an argument and returning a reference to part of it means that the lifetime of the returned reference is related to the lifetime of the `post` argument.

And we're done—all of Listing 17-11 now works! We've implemented the state pattern for the blog post workflow. The logic related to the rules lives in the state objects scattered throughout `Post`.

Trade-offs of the State Pattern

We've shown that Rust is capable of implementing the object-oriented state pattern. The different kinds of behavior a post should have in each state. The methods implement nothing about the various behaviors. The way we organized the code, we have to place to know the different ways a published post can behave: the implementations of the trait on the `Published` struct.

If we were to create an alternative implementation that didn't use the state pattern, instead use `match` expressions in the methods on `Post` or even in the `main` function. This would mean we have to go to several places to understand all the implications of a post being in the published state. Not only increase the more states we added: each of those `match` expressions would have to change.

With the state pattern, the `Post` methods and the places we use `Post` don't have to change. To add a new state, we would only need to add a new struct and implement the trait methods on that one struct.

The implementation using the state pattern is easy to extend to add more functionality. For the simplicity of maintaining code that uses the state pattern, try a few of these:

- Add a `reject` method that changes the post's state from `PendingReview` to `Draft`.
- Require two calls to `approve` before the state can be changed to `Published`.
- Allow users to add text content only when a post is in the `Draft` state. This makes the object responsible for what might change about the content but not responsible for the `Post`.

One downside of the state pattern is that, because the states implement the `State` trait, some of the states are coupled to each other. If we add another state like `PendingReview` and `Published`, such as `Scheduled`, we would have to change the logic for `PendingReview` to transition to `Scheduled` instead. It would be less work if we just added a new state, but that would mean switching between states in the pattern.

Another downside is that we've duplicated some logic. To eliminate some of the duplication, we might try to make default implementations for the `request_review` and `apply_review` methods on `Post`. Both methods delegate to the implementation of the `State` trait that return `self`; however, this would violate object safety, because we don't know what the concrete `self` will be exactly. We want to be able to use `state` methods on `Post` without needing its methods to be object safe.

Other duplication includes the similar implementations of the `request_review` and `apply_review` methods on `Post`. Both methods delegate to the implementation of the `State` trait in the `state` field of `Option` and set the new value of the `state` field to the new value. Since the methods on `Post` that followed this pattern, we might consider defining a new macro (see Appendix D for more on macros).

By implementing the state pattern exactly as it's defined for object-oriented taking as full advantage of Rust's strengths as we could. Let's look at some code from the `blog` crate that can make invalid states and transitions into compile time errors.

Encoding States and Behavior as Types

We'll show you how to rethink the state pattern to get a different set of trade-offs by encapsulating the states and transitions completely so outside code has no knowledge of them. Instead, we'll encode the states into different types. Consequently, Rust's type checking system will catch errors before they happen. For example, if we attempt to use `add_text` on a `Post` that attempts to use draft posts where only published posts are allowed by issuing a warning.

Let's consider the first part of `main` in Listing 17-11:

Filename: `src/main.rs`

```
fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");
    assert_eq!("", post.content());
}
```

We still enable the creation of new posts in the draft state using `Post::new`, but instead of having a `content` method on a `DraftPost` that returns an empty string, we'll make it so draft posts don't have the `content` method at all. When we try to call `content` on a `DraftPost`, we'll get a compiler error telling us the method does not exist. This makes it impossible for us to accidentally display draft post content in production because the code won't even compile. Listing 17-19 shows the definition of a `Post` struct and `DraftPost` struct, as well as methods on each:

Filename: `src/lib.rs`

```
pub struct Post {
    content: String,
}

pub struct DraftPost {
    content: String,
}

impl Post {
    pub fn new() -> DraftPost {
        DraftPost {
            content: String::new(),
        }
    }

    pub fn content(&self) -> &str {
        &self.content
    }
}

impl DraftPost {
    pub fn add_text(&mut self, text: &str) {
        self.content.push_str(text);
    }
}
```

Listing 17-19: A `Post` with a `content` method and a `DraftPost` without a `content` method.

Both the `Post` and `DraftPost` structs have a private `content` field that stores the post's text.

The structs no longer have the `state` field because we're moving the encoding logic into the `Post` struct. The `DraftPost` struct will represent a published post, and it has a `content` field that returns the `content`.

We still have a `Post::new` function, but instead of returning an instance of `DraftPost`, it returns an instance of `DraftPost`. Because `content` is private and there aren't any functions that can access it, it's not possible to create an instance of `Post` right now.

The `DraftPost` struct has an `add_text` method, so we can add text to `content`. However, `DraftPost` does not have a `content` method defined! So now the programmatic constraint that `DraftPost` can only be created from `Post::new` and `DraftPost` can only be created from `Post::new` starts to break. If we try to start as draft posts, and draft posts don't have their content available for distribution, then trying to add text to them will result in a compiler error.

Implementing Transitions as Transformations into Different Types

So how do we get a published post? We want to enforce the rule that a draft post can only be created from a `Post`, and that it must be approved before it can be published. A post in the pending review state can only be created from a `DraftPost`, and it must have any content. Let's implement these constraints by adding another struct, `PendingReviewPost`, and defining the `request_review` method on `DraftPost` to return a `PendingReviewPost`. We'll also implement an `approve` method on `PendingReviewPost` to return a `Post`, as shown in Listing 17-20.

Filename: `src/lib.rs`

```
impl DraftPost {
    // --snip--

    pub fn request_review(self) -> PendingReviewPost {
        PendingReviewPost {
            content: self.content,
        }
    }
}

pub struct PendingReviewPost {
    content: String,
}

impl PendingReviewPost {
    pub fn approve(self) -> Post {
        Post {
            content: self.content,
        }
    }
}
```

Listing 17-20: A `PendingReviewPost` that gets created by calling `request_review` on a `DraftPost`, and an `approve` method that turns a `PendingReviewPost` into a published `Post`

The `request_review` and `approve` methods take ownership of `self`, thus creating new `PendingReviewPost` and `Post` instances and transforming them into owned values. This way, we won't have any lingering references to the old `DraftPost` or `PendingReviewPost` instances after we've called `request_review` on them, and so forth. The `PendingReviewPost` struct does not have a `content` method defined on it, so attempting to read its content results in a compile error. This is similar to how we handled the `DraftPost`. Because the only way to get a published `Post` instance is to call the `approve` method on a `PendingReviewPost`, the only way to get a `PendingReviewPost` is to call the `request_review` method on a `DraftPost`. This enforces the blog post workflow into the type system.

But we also have to make some small changes to `main`. The `request_review` method needs to return a `PendingReviewPost` instead of a `DraftPost`.

methods return new instances rather than modifying the struct they're called on. This means more `let` post = shadowing assignments to save the returned instances. You can't make assertions about the draft and pending review posts's contents be empty strings because we can't compile code that tries to use the content of posts in those states. The updated code in `main` is shown in Listing 17-21:

Filename: `src/main.rs`

```
extern crate blog;
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salad for lunch today");

    let post = post.request_review();

    let post = post.approve();

    assert_eq!("I ate a salad for lunch today", post.content());
}
```

Listing 17-21: Modifications to `main` to use the new implementation of the `Post` trait.

The changes we needed to make to `main` to reassign `post` mean that this implementation of the `Post` trait no longer follows the object-oriented state pattern anymore: the transformations are now impossible because of the type system and the type checking happens at compile time! This ensures that certain bugs, such as displaying the content of a post before it has been approved, will be discovered before they make it to production.

Try the tasks suggested for additional requirements that we mentioned at the end of the `blog` crate as it is after Listing 17-20 to see what you think about the design of the code. Note that some of the tasks might be completed already in this chapter.

We've seen that even though Rust is capable of implementing object-oriented patterns, such as encoding state into the type system, are also available in Rust. These different trade-offs. Although you might be very familiar with object-oriented patterns, the problem to take advantage of Rust's features can provide benefits, such as catching bugs at compile time. Object-oriented patterns won't always be the best solution for certain features, like ownership, that object-oriented languages don't have.

Summary

No matter whether or not you think Rust is an object-oriented language after reading this chapter, you now know that you can use trait objects to get some object-oriented features. Trait dispatch can give your code some flexibility in exchange for a bit of runtime overhead. You can use this flexibility to implement object-oriented patterns that can help your code be more readable. Rust also has other features, like ownership, that object-oriented languages don't have. An object-oriented pattern won't always be the best way to take advantage of Rust's strengths, so it's good to know that there are other available options.

Next, we'll look at patterns, which are another of Rust's features that enable you to reuse code. You may have looked at them briefly throughout the book but haven't seen their full capabilities yet.

Patterns and Matching

Patterns are a special syntax in Rust for matching against the structure of types. Patterns are simple. Using patterns in conjunction with `match` expressions and other control structures gives you more control over a program's control flow. A pattern consists of some combinators:

- Literals
- Destructured arrays, enums, structs, or tuples
- Variables
- Wildcards
- Placeholders

These components describe the shape of the data we're working with, which allows us to determine whether our program has the correct data to continue running a piece of code.

To use a pattern, we compare it to some value. If the pattern matches the value, the code in that part of our code will run. Recall the `match` expressions in Chapter 6 that used patterns to filter out sorting machine example. If the value fits the shape of the pattern, we can use it; if it doesn't, the code associated with the pattern won't run.

This chapter is a reference on all things related to patterns. We'll cover the various kinds of patterns, the difference between refutable and irrefutable patterns, and the various ways to use patterns. By the end of the chapter, you'll know how to express many concepts in a clear way.

All the Places Patterns Can Be Used

Patterns pop up in a number of places in Rust, and you've been using them without realizing it. This section discusses all the places where patterns are valid.

match Arms

As discussed in Chapter 6, we use patterns in the arms of `match` expression. The arms of a `match` expression are defined as the keyword `match`, a value to match on, and one or more patterns. Each arm consists of a pattern and an expression to run if the value matches that pattern:

```
match VALUE {  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
    PATTERN => EXPRESSION,  
}
```

One requirement for `match` expressions is that they need to be *exhaustive* in that all possibilities for the value in the `match` expression must be accounted for. One way to ensure that every possibility is covered is to have a catchall pattern for the last arm: for example, `match x { ... _ => ... }`. This pattern matches any value and thus covers every remaining case.

A particular pattern `_` will match anything, but it never binds to a variable, so it's only useful in the last match arm. The `_` pattern can be useful when you want to ignore any value that doesn't match any other pattern. For example, we'll cover the `_` pattern in more detail in the "Ignoring Values in `match`" section of this chapter.

Conditional `if let` Expressions

In Chapter 6 we discussed how to use `if let` expressions mainly as a short-hand way to handle optional values.

equivalent of a `match` that only matches one case. Optionally, `if let` can have an `else` containing code to run if the pattern in the `if let` doesn't match.

Listing 18-1 shows that it's also possible to mix and match `if let`, `else if`, and `else` expressions. Doing so gives us more flexibility than a `match` expression in which we can only one value to compare with the patterns. Also, the conditions in a series of `else if let` arms aren't required to relate to each other.

The code in Listing 18-1 shows a series of checks for several conditions that determine what background color should be. For this example, we've created variables with likely values that a real program might receive from user input.

Filename: `src/main.rs`

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {}, as the background");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

Listing 18-1: Mixing `if let`, `else if`, `else if let`, and `else`

If the user specifies a favorite color, that color is the background color. If today is Tuesday, the background color is green. If the user specifies their age as a string and we can parse it successfully, the color is either purple or orange depending on the value of the age. If none of these conditions apply, the background color is blue.

This conditional structure lets us support complex requirements. With the help of `if let`, we can have here, this example will print `Using purple as the background color`.

You can see that `if let` can also introduce shadowed variables in the same scope. In Listing 18-1, the line `if let Ok(age) = age` introduces a new shadowed `age` variable inside the `Ok` variant. This means we need to place the `if age > 30` condition inside the `Ok` block: we can't combine these two conditions into `if let Ok(age) = age & age > 30` because the shadowed `age` we want to compare to 30 isn't valid until the new scope starts after the `Ok` bracket.

The downside of using `if let` expressions is that the compiler doesn't check for errors in the conditions like it does with `match` expressions. If we omitted the last `else` block from Listing 18-1 and tried to handle some cases, the compiler would not alert us to the possible logic bug.

while let Conditional Loops

Similar in construction to `if let`, the `while let` conditional loop allows a loop to continue as long as a pattern continues to match. The example in Listing 18-2 shows a way to reverse a vector as a stack and prints the values in the vector in the opposite order it was pushed onto the stack.

pushed.

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

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{}", top);
}
```

Listing 18-2: Using a `while let` loop to print values for as long as `stack.pop()` returns `Some(value)`.

This example prints 3, 2, and then 1. The `pop` method takes the last element off the vector and returns `Some(value)`. If the vector is empty, `pop` returns `None`. The `while` loop continues to run the code in its block as long as `pop` returns `Some`. When `pop` returns `None`, the loop ends. We can use `while let` to pop every element off our stack.

for Loops

In Chapter 3, we mentioned that the `for` loop is the most common loop construct in Rust. In fact, we haven't yet discussed the pattern that `for` takes. In a `for` loop, the pattern follows the keyword `for`, so in `for x in y` the `x` is the pattern.

Listing 18-3 demonstrates how to use a pattern in a `for` loop to destructure a tuple as part of the `for` loop.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{} is at index {}", value, index);
}
```

Listing 18-3: Using a pattern in a `for` loop to destructure a tuple.

The code in Listing 18-3 will print the following:

```
a is at index 0
b is at index 1
c is at index 2
```

We use the `enumerate` method to adapt an iterator to produce a value and its index as a tuple. The first call to `enumerate` produces the tuple `(0, "a")`. This tuple is matched to the pattern `(index, value)`, `index` will be `0` and `value` will be `"a"`, which is printed as the first line of the output.

let Statements

Prior to this chapter, we had only explicitly discussed using patterns with `match`. In fact, we've used patterns in other places as well, including in `let` statements. Listing 18-4 shows this straightforward variable assignment with `let`:

```
let x = 5;
```

Throughout this book, we've used `let` like this hundreds of times, and although you might not have realized it, you were using patterns! More formally, a `let` statement looks like this:

```
let PATTERN = EXPRESSION;
```

In statements like `let x = 5;` with a variable name in the `PATTERN` slot, the pattern is particularly simple form of a pattern. Rust compares the expression against any names it finds. So in the `let x = 5;` example, `x` is a pattern that means "the value `5` bound to the variable `x`." Because the name `x` is the whole pattern, this pattern matches any value that has the same type as `5`.

To see the pattern matching aspect of `let` more clearly, consider Listing 18-4, which shows how to destructure a tuple.

```
let (x, y, z) = (1, 2, 3);
```

Listing 18-4: Using a pattern to destructure a tuple and create three variables

Here, we match a tuple against a pattern. Rust compares the value `(1, 2, 3)` against the pattern `(x, y, z)` and sees that the value matches the pattern, so Rust binds `1` to `x`, `2` to `y`, and `3` to `z`. You can think of this tuple pattern as nesting three individual variable patterns.

If the number of elements in the pattern doesn't match the number of elements in the tuple, the overall type won't match and we'll get a compiler error. For example, Listing 18-5 shows how to destructure a tuple with three elements into two variables, which won't work:

```
let (x, y) = (1, 2, 3);
```

Listing 18-5: Incorrectly constructing a pattern whose variables don't match the number of elements in the tuple

Attempting to compile this code results in this type error:

```
error[E0308]: mismatched types
--> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |           ^^^^^^ expected a tuple with 3 elements, found one with 2
  |
= note: expected type `({integer}, {integer}, {integer})`
        found type `(_, _)`
```

If we wanted to ignore one or more of the values in the tuple, we could use the "Ignoring Values in a Pattern" section. If the problem is that we have too many variables in the pattern, the solution is to make the types match by removing variables so that the pattern has the same number of elements as the tuple. For example, if we wanted to ignore the third element of the tuple in Listing 18-5, we could write:

Function Parameters

Function parameters can also be patterns. The code in Listing 18-6, which defines a function `foo` that takes one parameter named `x` of type `i32`, should by now look familiar:

```
fn foo(x: i32) {
    // code goes here
}
```

Listing 18-6: A function signature uses patterns in the parameters

The `x` part is a pattern! As we did with `let`, we could match a tuple in a function pattern. Listing 18-7 splits the values in a tuple as we pass it to a function.

Filename: `src/main.rs`

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({}, {})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Listing 18-7: A function with parameters that destructure a tuple

This code prints `current location: (3, 5)`. The values `&(3, 5)` match the pattern `(x, y)`, where `x` is the value `3` and `y` is the value `5`.

We can also use patterns in closure parameter lists in the same way as in functions because closures are similar to functions, as discussed in Chapter 13.

At this point, you've seen several ways of using patterns, but patterns don't always place we can use them. In some places, the patterns must be irrefutable; in others, they can be refutable. We'll discuss these two concepts next.

Refutability: Whether a Pattern Might Fail to Match

Patterns come in two forms: refutable and irrefutable. Patterns that will match anything passed are *irrefutable*. An example would be `x` in the statement `let x = 5;` because `x` can match anything and therefore cannot fail to match. Patterns that can fail to match if they don't match the value are *refutable*. An example would be `Some(x)` in the expression `if let Some(x) = a_value`. If `a_value` is `None`, the `Some(x)` pattern will not match.

Function parameters, `let` statements, and `for` loops can only accept irrefutable patterns; the program cannot do anything meaningful when values don't match. The `if` expression only accepts refutable patterns, because by definition they're intended to indicate possible failure: the functionality of a conditional is in its ability to perform different actions based on success or failure.

In general, you shouldn't have to worry about the distinction between refutable and irrefutable patterns; however, you do need to be familiar with the concept of refutability when you see it in an error message. In those cases, you'll need to change either the pattern you're using or the construct you're using the pattern with, depending on the intended behavior.

Let's look at an example of what happens when we try to use a refutable pattern when it requires an irrefutable pattern and vice versa. Listing 18-8 shows a `let` statement with a pattern we've specified `Some(x)`, a refutable pattern. As you might expect, the code won't compile.

```
let Some(x) = some_option_value;
```

Listing 18-8: Attempting to use a refutable pattern with `let`

If `some_option_value` was a `None` value, it would fail to match the pattern `Some(x)`. This pattern is refutable. However, the `let` statement can only accept an irrefutable pattern.

there is nothing valid the code can do with a `None` value. At compile time, Rust has tried to use a refutable pattern where an irrefutable pattern is required.

```
error[E0005]: refutable pattern in local binding: `None` not covered
-->
|
3 | let Some(x) = some_option_value;
|     ^^^^^^ pattern `None` not covered
```

Because we didn't cover (and couldn't cover!) every valid value with the pattern, the compiler rightfully produces a compiler error.

To fix the problem where we have a refutable pattern where an irrefutable pattern is required, we can change the code that uses the pattern: instead of using `let`, we can use `if`. If the pattern doesn't match, the code will just skip the code in the curly brackets, and continue validly. Listing 18-9 shows how to fix the code in Listing 18-8.

```
if let Some(x) = some_option_value {
    println!("{}", x);
}
```

Listing 18-9: Using `if let` and a block with refutable patterns instead of `let`

We've given the code an `out!` This code is perfectly valid, although it means we're using a refutable pattern without receiving an error. If we give `if let` a pattern that doesn't match, such as `x`, as shown in Listing 18-10, it will not compile.

```
if let x = 5 {
    println!("{}", x);
};
```

Listing 18-10: Attempting to use an irrefutable pattern with `if let`

Rust complains that it doesn't make sense to use `if let` with an irrefutable pattern:

```
error[E0162]: irrefutable if-let pattern
--> <anon>:2:8
|
2 | if let x = 5 {
|     ^ irrefutable pattern
```

For this reason, match arms must use refutable patterns, except for the last match arm, which allows us to use an irrefutable pattern to match any remaining values with an irrefutable pattern. Rust allows us to use `match` with only one arm, but this syntax isn't particularly useful and could be replaced by a simpler `let` statement.

Now that you know where to use patterns and the difference between refutable and irrefutable patterns, let's cover all the syntax we can use to create patterns.

Pattern Syntax

Throughout the book, you've seen examples of many kinds of patterns. In this chapter, we'll go over the syntax valid in patterns and discuss why you might want to use each one.

Matching Literals

As you saw in Chapter 6, you can match patterns against literals directly. The some examples:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one` because the value in `x` is 1. This syntax is useful when take an action if it gets a particular concrete value.

Matching Named Variables

Named variables are irrefutable patterns that match any value, and we've us the book. However, there is a complication when you use named variables ir Because `match` starts a new scope, variables declared as part of a pattern ir expression will shadow those with the same name outside the `match` consti all variables. In Listing 18-11, we declare a variable named `x` with the value `y` with the value `10`. We then create a `match` expression on the value `x`. L the match arms and `println!` at the end, and try to figure out what the coc running this code or reading further.

Filename: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {:?}", y),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

Listing 18-11: A `match` expression with an arm that introduces a shadowed variable.

Let's walk through what happens when the `match` expression runs. The pattern arm doesn't match the defined value of `x`, so the code continues.

The pattern in the second match arm introduces a new variable named `y` that inside a `Some` value. Because we're in a new scope inside the `match` expression, not the `y` we declared at the beginning with the value 10. This new `y` matches the value `5` inside the `Some` value of `x`. Therefore, this new `y` is used in the `Matched, y = ...` part of the `Some` in `x`. That value is `5`, so the expression for that arm executes a `println!` with the value `5`.

If `x` had been a `None` value instead of `Some(5)`, the patterns in the first two matched, so the value would have matched to the underscore. We didn't introduce the pattern of the underscore arm, so the `x` in the expression is still the outer variable and not shadowed. In this hypothetical case, the `match` would print `Default case,`

When the `match` expression is done, its scope ends, and so does the scope of `println!` produces at the end: `x = Some(5), y = 10`.

To create a `match` expression that compares the values of the outer `x` and introducing a shadowed variable, we would need to use a match guard condition about match guards later in the "Extra Conditionals with Match Guards" section.

Multiple Patterns

In `match` expressions, you can match multiple patterns using the `|` syntax, for example, the following code matches the value of `x` against the match arms: an `or` option, meaning if the value of `x` matches either of the values in that arm, it will run:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one or two`.

Matching Ranges of Values with ...

The `...` syntax allows us to match to an inclusive range of values. In the following code, the pattern `1 ... 5` matches any of the values within the range, that arm will execute:

```
let x = 5;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("something else"),
}
```

If `x` is 1, 2, 3, 4, or 5, the first arm will match. This syntax is more convenient than using multiple `||` operators to express the same idea; instead of `1 || 2 || 3 || 4 || 5`, we would have to specify `1 | 2 | 3 | 4 | 5` if we used `|`. Specifying a range is much shorter, especially when matching over a large range, say, any number between 1 and 1,000!

Ranges are only allowed with numeric values or `char` values, because the compiler needs to know at compile time whether the range is empty or not. The range isn't empty at compile time. The only types for which Rust can tell if a range is empty are `char` and numeric values.

Here is an example using ranges of `char` values:

```
let x = 'c';

match x {
    'a' ... 'j' => println!("early ASCII letter"),
    'k' ... 'z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Rust can tell that `c` is within the first pattern's range and prints early ASCII.

Destructuring to Break Apart Values

We can also use patterns to destructure structs, enums, tuples, and references of these values. Let's walk through each value.

Destructuring Structs

Listing 18-12 shows a `Point` struct with two fields, `x` and `y`, that we can break apart with a `let` statement.

Filename: `src/main.rs`

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let p = Point { x: 0, y: 7 };  
  
    let Point { x: a, y: b } = p;  
    assert_eq!(0, a);  
    assert_eq!(7, b);  
}
```

Listing 18-12: Destructuring a struct's fields into separate variables

This code creates the variables `a` and `b` that match the values of the `x` and `y` field of the `Point` variable. This example shows that the names of the variables in the pattern don't have to match the field names of the struct. But it's common to want the variable names to match the field names to make it easier to remember which variables came from which fields.

Because having variable names match the fields is common and because writing `let Point { x: x, y: y } = p;` contains a lot of duplication, there is a shorthand for matching struct fields: you only need to list the name of the struct field, and the variable created in the `let` pattern will have the same name. Listing 18-13 shows code that behaves like Listing 18-12, but the variables created in the `let` pattern are `x` and `y`.

Filename: `src/main.rs`

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let p = Point { x: 0, y: 7 };  
  
    let Point { x, y } = p;  
    assert_eq!(0, x);  
    assert_eq!(7, y);  
}
```

Listing 18-13: Destructuring struct fields using struct field shorthand

This code creates the variables `x` and `y` that match the `x` and `y` fields of the `Point` struct. The outcome is that the variables `x` and `y` contain the values from the `p` struct.

We can also destructure with literal values as part of the struct pattern rather than for all the fields. Doing so allows us to test some of the fields for particular values and then use those variables to destructure the other fields.

Listing 18-14 shows a `match` expression that separates `Point` values into three categories: directly on the `x` axis (which is true when `y = 0`), on the `y` axis (`x = 0`), or neither.

Filename: `src/main.rs`

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {}", x),
        Point { x: 0, y } => println!("On the y axis at {}", y),
        Point { x, y } => println!("On neither axis: ({}, {})", x, y)
    }
}
```

Listing 18-14: Destructuring and matching literal values in one pattern

The first arm will match any point that lies on the `x` axis by specifying that the value of the `y` field matches the literal `0`. The pattern still creates an `x` variable that we can use in the `println!` statement.

Similarly, the second arm matches any point on the `y` axis by specifying that its value is `0` and creates a variable `y` for the value of the `y` field. The third arm matches any other `Point` and creates variables for both the `x` and `y` fields.

In this example, the value `p` matches the second arm by virtue of `x` containing `0`, so it prints `On the y axis at 7`.

Destructuring Enums

We've destructured enums earlier in this book, for example, when we deconstructed the `Message` enum in Listing 6-5 in Chapter 6. One detail we haven't mentioned explicitly is that the variants of an enum should correspond to the way the data stored within the enum is deconstructed. In Listing 18-15 we use the `Message` enum from Listing 6-2 and write a `match` expression to destructure each inner value.

Filename: `src/main.rs`

```

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

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.");
        },
        Message::Move { x, y } => {
            println!(
                "Move in the x direction {} and in the y direction {}",
                x,
                y
            );
        }
        Message::Write(text) => println!("Text message: {}", text);
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}

```

Listing 18-15: Destructuring enum variants that hold different kinds of value:

This code will print `Change the color to red 0, green 160, and blue 255` if `msg` is `ChangeColor(0, 160, 255)`.

For enum variants without any data, like `Message::Quit`, we can't destructure them. We can only match on the literal `Message::Quit` value, and no variables are created.

For struct-like enum variants, such as `Message::Move`, we can use a pattern to match structs. After the variant name, we place curly brackets and variables so we break apart the pieces to use in the code for this arm. Here's how it would look for the `Move` variant as we did in Listing 18-13.

For tuple-like enum variants, like `Message::Write` that holds a tuple with one variable, or `Message::ChangeColor` that holds a tuple with three elements, the pattern is similar. We specify to match tuples. The number of variables in the pattern must match the number of elements in the variant we're matching.

Destructuring Nested Structs & Enums

Up until now, all of our examples have been matching structures that were completely flat. But what if we want to work with nested structures too?

We can refactor the example above to support both RGB and HSV colors:

```

enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32)
}

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

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        },
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!(
                "Change the color to hue {}, saturation {}, and va",
                h,
                s,
                v
            )
        }
        _ => {}
    }
}

```

Destructuring References

When the value we're matching to our pattern contains a reference, we need a reference from the value, which we can do by specifying a `&` in the pattern. This variable holding the value that the reference points to rather than getting a reference. This technique is especially useful in closures where we have iter references, but we want to use the values in the closure rather than the references.

The example in Listing 18-16 iterates over references to `Point` instances in the reference and the struct so we can perform calculations on the `x` and `y`.

```

let points = vec![
    Point { x: 0, y: 0 },
    Point { x: 1, y: 5 },
    Point { x: 10, y: -3 },
];
let sum_of_squares: i32 = points
    .iter()
    .map(|&Point { x, y }| x * x + y * y)
    .sum();

```

Listing 18-16: Destructuring a reference to a struct into the struct field values

This code gives us the variable `sum_of_squares` holding the value 135, which is the sum of the squares of the `x` value and the `y` value, adding those together, and then adding them up for all the `Point` instances in the `points` vector to get one number.

If we had not included the `&` in `&Point { x, y }`, we'd get a type mismatch. If we did, the code would then iterate over references to the items in the vector rather than the values themselves. This would look like this:

```
error[E0308]: mismatched types
-->
|
14 |         .map(|Point { x, y }| x * x + y * y)
|                         ^^^^^^^^^^ expected &Point, found struct `Point`
|
= note: expected type `&Point`
        found type `Point`
```

This error indicates that Rust was expecting our closure to match `&Point`, but instead it was able to bind directly to a `Point` value, not a reference to a `Point`.

Destructuring Structs and Tuples

We can mix, match, and nest destructuring patterns in even more complex ways. For example, this example shows a complicated destructure where we nest structs and tuples to extract all the primitive values out:

```
let ((feet, inches), Point {x, y}) = ((3, 10), Point { x: 3, y: -10 })
```

This code lets us break complex types into their component parts so we can work with them separately.

Destructuring with patterns is a convenient way to use pieces of values, such as individual fields in a struct, separately from each other.

Ignoring Values in a Pattern

You've seen that it's sometimes useful to ignore values in a pattern, such as `_`. For example, if you want to match on a tuple but don't care about the first value, you can use `_` to get a catchall that doesn't actually do anything but does account for the value. There are a few ways to ignore entire values or parts of values in a pattern: using the `_` pattern (which you've seen), using the `_` pattern within another pattern, using the `..` pattern with an underscore, or using `..` to ignore remaining parts of a value. Let's explore each of these patterns.

Ignoring an Entire Value with `_`

We've used the underscore (`_`) as a wildcard pattern that will match any value. Although the underscore `_` pattern is especially useful as the last argument in a function signature, we can use it in any pattern, including function parameters, as shown in Listing 18-17.

Filename: `src/main.rs`

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {}", y);
}

fn main() {
    foo(3, 4);
}
```

Listing 18-17: Using `_` in a function signature

```
This code will completely ignore the value passed as the first argument, 3, and
This code only uses the y parameter: 4.
```

In most cases when you no longer need a particular function parameter, you can simply ignore it by giving it an underscore. This is especially useful in some cases, for example, when implementing a trait where the function has a specific type signature but the function body in your implementation doesn't need to match it. The compiler will then not warn about unused function parameters, as it won't be able to use them instead.

Ignoring Parts of a Value with a Nested `_`

We can also use `_` inside another pattern to ignore just part of a value, for example to test for only part of a value but have no use for the other parts in the corresponding pattern. Listing 18-18 shows code responsible for managing a setting's value. The requirements are that the user should not be allowed to overwrite an existing setting but can unset the setting and can give the setting a value if it is currently set.

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

Listing 18-18: Using an underscore within patterns that match `Some` variants to ignore the value inside the `Some`.

This code will print `Can't overwrite an existing customized value` and the value `setting is Some(5)`. In the first match arm, we don't need to match on or ignore either `Some` variant, but we do need to test for the case when `setting_value` and `new_setting_value` are the `Some` variant. In that case, we print why we're not changing `setting_value`, and it doesn't get changed.

In all other cases (if either `setting_value` or `new_setting_value` are `None`), in the second arm, we want to allow `new_setting_value` to become the value of `setting_value`.

We can also use underscores in multiple places within one pattern to ignore multiple values. Listing 18-19 shows an example of ignoring the second and fourth values in a tuple.

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {}, {}, {}", first, third, fifth)
    }
}
```

Listing 18-19: Ignoring multiple parts of a tuple

This code will print `Some numbers: 2, 8, 32`, and the values 4 and 16 will be ignored.

Ignoring an Unused Variable by Starting Its Name with _

If you create a variable but don't use it anywhere, Rust will usually issue a warning. This could be a bug. But sometimes it's useful to create a variable you won't use just for prototyping or just starting a project. In this situation, you can tell Rust not to warn about an unused variable by starting the name of the variable with an underscore. In Listing 18-20, we have two unused variables, but when we run this code, we should only get a warning.

Filename: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Listing 18-20: Starting a variable name with an underscore to avoid getting a warning

Here we get a warning about not using the variable `y`, but we don't get a warning about the variable preceded by the underscore.

Note that there is a subtle difference between using only `_` and using a name preceded by an underscore. The syntax `_x` still binds the value to the variable, whereas `_c` does not. Listing 18-21 shows a case where this distinction matters; Listing 18-21 will provide us with

```
let s = Some(String::from("Hello!"));

if let Some(_s) = s {
    println!("found a string");
}

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

Listing 18-21: An unused variable starting with an underscore still binds the value and retains ownership of the value

We'll receive an error because the `s` value will still be moved into `_s`, which is again bound to the value. However, using the underscore by itself doesn't ever bind to the value. Listing 18-22 shows the same code compile without any errors because `s` doesn't get moved into `_s`.

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

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

Listing 18-22: Using an underscore does not bind the value

This code works just fine because we never bind `s` to anything; it isn't moved into `_s`.

Ignoring Remaining Parts of a Value with ..

With values that have many parts, we can use the `..` syntax to use only a few parts of the value, avoiding the need to list underscores for each ignored value. The `..` pattern matches the rest of a value that we haven't explicitly matched in the rest of the pattern. In Listing 18-23, we have a `Point` struct that holds a coordinate in three-dimensional space. In the `map` function, we want to operate only on the `x` coordinate and ignore the values in the `y` and `z` fields.

```

struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}

```

Listing 18-23: Ignoring all fields of a `Point` except for `x` by using `..`

We list the `x` value and then just include the `..` pattern. This is quicker than listing every field like `y: _, z: _`, particularly when we're working with structs that have lots of fields and only one or two fields are relevant.

The syntax `..` will expand to as many values as it needs to be. Listing 18-24 shows how to ignore everything in a tuple with a tuple.

Filename: `src/main.rs`

```

fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {}, {}", first, last);
        },
    }
}

```

Listing 18-24: Matching only the first and last values in a tuple and ignoring everything in between

In this code, the first and last value are matched with `first` and `last`. The `..` pattern ignores everything in the middle.

However, using `..` must be unambiguous. If it is unclear which values are included in the `..` range and which should be ignored, Rust will give us an error. Listing 18-25 shows an example where the range is ambiguous, so it will not compile.

Filename: `src/main.rs`

```

fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {}", second)
        },
    }
}

```

Listing 18-25: An attempt to use `..` in an ambiguous way

When we compile this example, we get this error:

```
error: `..` can only be used once per tuple or tuple struct pattern
--> src/main.rs:5:22
 |
5 |         (.., second, ..) => {
|             ^^
```

It's impossible for Rust to determine how many values in the tuple to ignore with `second` and then how many further values to ignore thereafter. This code wants to ignore 2, bind `second` to 4, and then ignore 8, 16, and 32; or the first 4, bind `second` to 8, and then ignore 16 and 32; and so forth. The `..` doesn't mean anything special to Rust, so we get a compiler error because the code like this is ambiguous.

Creating References in Patterns with `ref` and `ref mut`

Let's look at using `ref` to make references so ownership of the values isn't transferred to the pattern. Usually, when you match against a pattern, the variables introduced by the pattern are bound to a value. Rust's ownership rules mean the value will be moved into the variable if you're using the pattern. Listing 18-26 shows an example of a `match` that has a variable and then usage of the entire value in the `println!` statement later. The code will fail to compile because ownership of part of the `robot_name` value has been moved into the `name` variable in the pattern of the first `match` arm.

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(name) => println!("Found a name: {}", name),
    None => (),
}

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

Listing 18-26: Creating a variable in a `match` arm pattern takes ownership of the variable.

Because ownership of part of `robot_name` has been moved to `name`, we can't use `robot_name` in the `println!` after the `match` because `robot_name` no longer exists.

To fix this code, we want to make the `Some(name)` pattern *borrow* that part of the value instead of taking ownership. You've already seen that, outside of patterns, the way to create a reference is to use `&`, so you might think the solution is changing `Some` to `Some(&name)`.

However, as you saw in the “Destructuring to Break Apart Values” section, the `Some` pattern does not *create* a reference but *matches* an existing reference in the value. Because of that meaning in patterns, we can't use `&` to create a reference in a pattern.

Instead, to create a reference in a pattern, we use the `ref` keyword before the `name` variable, as shown in Listing 18-27.

```
let robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref name) => println!("Found a name: {}", name),
    None => (),
}

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

Listing 18-27: Creating a reference so a pattern variable does not take ownership

This example will compile because the value in the `Some` variant in `robot_name` is consumed by the `match`; the `match` only took a reference to the data in `robot_name` rather than taking ownership.

To create a mutable reference so we're able to mutate a value matched in a pattern, use `ref mut` instead of `&mut`. The reason is, again, that in patterns, the latter is creating new references, not creating mutable references, not creating new ones. Listing 18-28 shows an example of how to create a mutable reference.

```
let mut robot_name = Some(String::from("Bors"));

match robot_name {
    Some(ref mut name) => *name = String::from("Another name"),
    None => (),
}

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

Listing 18-28: Creating a mutable reference to a value as part of a pattern usage

This example will compile and print `robot_name is: Some("Another name")`. Because we're creating a mutable reference, we need to dereference within the match arm code using `*name` to actually mutate the value.

Extra Conditionals with Match Guards

A *match guard* is an additional `if` condition specified after the pattern in a `match` expression. This allows the pattern to also match, along with the pattern matching, for that arm to be chosen. Match guards allow us to express more complex ideas than a pattern alone allows.

The condition can use variables created in the pattern. Listing 18-29 shows a `match` expression where the first arm has the pattern `Some(x)` and also has a match guard of `if x < 5`.

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}!", x),
    None => (),
}
```

Listing 18-29: Adding a match guard to a pattern

This example will print `less than five: 4`. When `num` is compared to the pattern `Some(x)`, it matches, because `Some(4)` matches `Some(x)`. Then the match guard checks that `x` is less than `5`, and because it is, the first arm is selected.

If `num` had been `Some(10)` instead, the match guard in the first arm would have failed because `10` is not less than `5`. Rust would then go to the second arm, which would match because the second arm doesn't have a match guard and therefore matches any `Some` variant.

There is no way to express the `if x < 5` condition within a pattern, so the `match` guard is the only way to express this logic.

In Listing 18-11, we mentioned that we could use match guards to solve our problem. Recall that a new variable was created inside the pattern in the `match` expression. That new variable meant we couldn't use the variable outside the `match`.

of the outer variable. Listing 18-30 shows how we can use a match guard to

Filename: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {:?}", n),
        _ => println!("Default case, x = {:?}", x),
    }

    println!("at the end: x = {:?}, y = {:?}", x, y);
}
```

Listing 18-30: Using a match guard to test for equality with an outer variable

This code will now print `Default case, x = Some(5)`. The pattern in the second arm introduce a new variable `y` that would shadow the outer `y`, meaning we can't use `y` in the match guard. Instead of specifying the pattern as `Some(y)`, which would have `y` shadow the outer `y`, we specify `Some(n)`. This creates a new variable `n` that doesn't shadow the outer `y`. There is no `n` variable outside the `match`.

The match guard `if n == y` is not a pattern and therefore doesn't introduce a new `y`. Instead, it is the outer `y` rather than a new shadowed `y`, and we can look for a value that matches the outer `y` by comparing `n` to `y`.

You can also use the `or` operator `|` in a match guard to specify multiple patterns. The condition will apply to all the patterns. Listing 18-31 shows the precedence of the match guard with a pattern that uses `|`. The important part of this example is that the `if` condition applies to `4`, `5`, *and* `6`, even though it might look like `if y` only applies to `6`.

```
let x = 4;
let y = false;

match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

Listing 18-31: Combining multiple patterns with a match guard

The match condition states that the arm only matches if the value of `x` is equal to `4`, `5`, or `6` and `y` is `true`. When this code runs, the pattern of the first arm matches because the condition `if y` is `false`, so the first arm is not chosen. The code moves on to the second arm, which does match, and this program prints `no`. The reason is that the `if` condition applies to the whole pattern `4 | 5 | 6`, not only to the last value `6`. In other words, the `if` condition in relation to a pattern behaves like this:

`(4 | 5 | 6) if y => ...`

rather than this:

`4 | 5 | (6 if y) => ...`

After running the code, the precedence behavior is evident: if the match guard applies to the final value in the list of values specified using the `|` operator, the arm will not run. In this case, the program would have printed `yes`.

Bindings

The `at` operator (`@`) lets us create a variable that holds a value at the same time as we test it. Listing 18-32 shows an example where we have a `Message::Hello` with an `id` field that is within the range `3...7`. But we also want to bind to the `id` value so we can use it in the code associated with the arm. The variable `id_variable` is the same as the field, but for this example we'll use a different name.

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3...7 } => {
        println!("Found an id in range: {}", id_variable)
    },
    Message::Hello { id: 10...12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```

Listing 18-32: Using `@` to bind to a value in a pattern while also testing it

This example will print `Found an id in range: 5`. By specifying `id_variable @ 3...7`, we're capturing whatever value matched the range while also testing the range pattern.

In the second arm, where we only have a range specified in the pattern, the arm doesn't have a variable that contains the actual value of the `id` field. This means that if the value had been 10, 11, or 12, but the code that goes with that pattern doesn't know the value. The pattern code isn't able to use the value from the `id` field, because we haven't bound it to a variable.

In the last arm, where we've specified a variable without a range, we do have access to its value. We can use it in the arm's code in a variable named `id`. The reason is that we've used the `id` variable in the shorthand syntax. But we haven't applied any test to the value in the `id` field. This means that both the first two arms: any value would match this pattern.

Using `@` lets us test a value and save it in a variable within one pattern.

Summary

Rust's patterns are very useful in that they help distinguish between different values. When used in `match` expressions, Rust ensures your patterns cover every possible value. Patterns in `let` statements and function parameters make them even more useful, enabling the destructuring of values into smaller parts at the same time as saving variables. We can create simple or complex patterns to suit our needs.

Next, for the penultimate chapter of the book, we'll look at some advanced aspects of Rust's features.

Advanced Features

By now, you've learned the most commonly used parts of the Rust programming language. In this chapter, we'll do one more project in Chapter 20, we'll look at a few aspects of the language that you might not use every once in a while. You can use this chapter as a reference for when you encounter situations where you need to use unknowns when using Rust. The features you'll learn to use in this chapter are not ones that you'll use all the time, but they're important to know about. Although you might not reach for them often, we want to make sure you have access to all the features Rust has to offer.

In this chapter, we'll cover:

- Unsafe Rust: how to opt out of some of Rust's guarantees and take responsibility for upholding those guarantees
- Advanced lifetimes: syntax for complex lifetime situations
- Advanced traits: associated types, default type parameters, fully qualified types, and the newtype pattern in relation to traits
- Advanced types: more about the newtype pattern, type aliases, the never type, and sized types
- Advanced functions and closures: function pointers and returning closures

It's a panoply of Rust features with something for everyone! Let's dive in!

Unsafe Rust

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time. However, Rust has a second language hidden inside it that doesn't enforce memory safety guarantees: it's called *unsafe Rust* and works just like regular Rust, but with superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler tries to determine whether or not code upholds the guarantees, it's better for it to reject programs rather than accept some invalid programs. Although the code might be safe, the compiler can't tell, so it's not! In these cases, you can use unsafe code to tell the compiler "I know what I'm doing." The downside is that you use it at your own risk: if you use it incorrectly, problems due to memory unsafety, such as null pointer dereferencing, can occur.

Another reason Rust has an unsafe alter ego is that the underlying compute system is unsafe. If Rust didn't let you do unsafe operations, you couldn't do certain tasks that require you to do low-level systems programming, such as directly interacting with the hardware or even writing your own operating system. Working with low-level systems is one of the goals of the language. Let's explore what we can do with unsafe Rust and how it can be useful.

Unsafe Superpowers

To switch to unsafe Rust, use the `unsafe` keyword and then start a new block of code. You can take four actions in unsafe Rust, called *unsafe superpowers*, that give you superpowers. Those superpowers include the ability to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

It's important to understand that `unsafe` doesn't turn off the borrow checker. It only lets you do things that the borrow checker can't catch.

Rust's safety checks: if you use a reference in unsafe code, it will still be checked. The `unsafe` keyword only gives you access to these four features that are then not checked for memory safety. You'll still get some degree of safety inside of an unsafe block.

In addition, `unsafe` does not mean the code inside the block is necessarily correct. You definitely have memory safety problems: the intent is that as the program runs, the code inside an `unsafe` block will access memory in a valid way.

People are fallible, and mistakes will happen, but by requiring these four unsafe features inside blocks annotated with `unsafe` you'll know that any errors related to raw pointers will be caught within an `unsafe` block. Keep `unsafe` blocks small; you'll be thankful later when you find memory bugs.

To isolate unsafe code as much as possible, it's best to enclose unsafe code in a safe abstraction and provide a safe API, which we'll discuss later in the chapter when we examine traits and methods. Parts of the standard library are implemented as safe abstractions that have been audited. Wrapping unsafe code in a safe abstraction prevents it from leaking out into all the places that you or your users might want to use the function. It's better to implement with `unsafe` code, because using a safe abstraction is safe.

Let's look at each of the four unsafe superpowers in turn. We'll also look at some ways to provide a safe interface to unsafe code.

Dereferencing a Raw Pointer

In Chapter 4, in the "Dangling References" section, we mentioned that the `as` operator guarantees that references are always valid. Unsafe Rust has two new types called *raw pointers*: `*const T` and `*mut T`, respectively. As with references, raw pointers can be immutable or mutable and can point to any type. The asterisk isn't the dereference operator; it's part of the pointer type name. In the context of raw pointers, *immutable* means that the pointer can't be modified after being dereferenced.

Different from references and smart pointers, raw pointers:

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers to the same location
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic cleanup

By opting out of having Rust enforce these guarantees, you can give up guarantees for greater performance or the ability to interface with another language where Rust's guarantees don't apply.

Listing 19-1 shows how to create an immutable and a mutable raw pointer from a reference.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: Creating raw pointers from references

Notice that we don't include the `unsafe` keyword in this code. We can create raw pointers from references in safe code; we just can't dereference raw pointers outside an unsafe block, as you'll see in the next section.

We've created raw pointers by using `as` to cast an immutable and a mutable reference to raw pointers.

corresponding raw pointer types. Because we created them directly from `ref`, they are valid, we know these particular raw pointers are valid, but we can't make just any raw pointer.

Next, we'll create a raw pointer whose validity we can't be so certain of. Listing 19-2 shows how to create a raw pointer to an arbitrary location in memory. Trying to use arbitrary raw pointers is undefined: there might be data at that address or there might not, the compiler won't know. If you write code so there is no memory access, or the program might error with a segmentation fault, there is no good reason to write code like this, but it is possible.

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: Creating a raw pointer to an arbitrary memory address

Recall that we can create raw pointers in safe code, but we can't *dereference* them to get the data being pointed to. In Listing 19-3, we use the dereference operator `*` to do this, which requires an `unsafe` block.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Listing 19-3: Dereferencing raw pointers within an `unsafe` block

Creating a pointer does no harm; it's only when we try to access the value that it might end up dealing with an invalid value.

Note also that in Listing 19-1 and 19-3, we created `*const i32` and `*mut i32` pointers that both pointed to the same memory location, where `num` is stored. If we instead tried to create an immutable and a mutable reference to `num`, the code would not have compiled because Rust's ownership rules don't allow a mutable reference at the same time as any immutable reference to the same memory location. With raw pointers, we can create a mutable pointer and an immutable pointer to the same memory location and change data through the mutable pointer, potentially creating a data race.

With all of these dangers, why would you ever use raw pointers? One major reason is interfacing with C code, as you'll see in the next section, "Calling an Unsafe Function or Method". Another case is when building up safe abstractions that the borrow checker can't handle. We'll introduce unsafe functions and then look at an example of a safe abstraction that uses them.

Calling an Unsafe Function or Method

The second type of operation that requires an unsafe block is calls to unsafe functions and methods. These look exactly like regular functions and methods, but are preceded by the `unsafe` keyword before the rest of the definition. The `unsafe` keyword in this context means that the function has requirements we need to uphold when we call this function, because Rust's ownership rules don't allow us to uphold them. By calling an unsafe function within an `unsafe` block, we've read this function's documentation and take responsibility for upholding its contracts.

Here is an unsafe function named `dangerous` that doesn't do anything in its

```
unsafe fn dangerous() {}  
unsafe {  
    dangerous();  
}
```

We must call the `dangerous` function within a separate `unsafe` block. If we without the `unsafe` block, we'll get an error:

```
error[E0133]: call to unsafe function requires unsafe function or I  
-->  
|  
4 |     dangerous();  
|     ^^^^^^^^^^ call to unsafe function
```

By inserting the `unsafe` block around our call to `dangerous`, we're asserting the function's documentation, we understand how to use it properly, and we fulfilling the contract of the function.

Bodies of unsafe functions are effectively `unsafe` blocks, so to perform other within an unsafe function, we don't need to add another `unsafe` block.

Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code doesn't mean we need to mark unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction let's study a function from the standard library, `split_at_mut`, that requires explore how we might implement it. This safe method is defined on `slice` and makes it two by splitting the slice at the index given as an argument. Listing use `split_at_mut`.

```
let mut v = vec![1, 2, 3, 4, 5, 6];  
let r = &mut v[..];  
let (a, b) = r.split_at_mut(3);  
assert_eq!(a, &mut [1, 2, 3]);  
assert_eq!(b, &mut [4, 5, 6]);
```

Listing 19-4: Using the safe `split_at_mut` function

We can't implement this function using only safe Rust. An attempt might look like Listing 19-5, which won't compile. For simplicity, we'll implement `split_at_mut` as a `slice` method and only for slices of `i32` values rather than for a generic type `T`.

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {  
    let len = slice.len();  
    assert!(mid <= len);  
    (&mut slice[..mid],  
     &mut slice[mid..])  
}
```

Listing 19-5: An attempted implementation of `split_at_mut` using only safe

This function first gets the total length of the slice. Then it asserts that the `in` parameter is within the slice by checking whether it's less than or equal to `len`. This means that if we pass an index that is greater than the `len` to split the slice, the function will panic before it attempts to use that index.

Then we return two mutable slices in a tuple: one from the start of the original slice to `mid`, and another from `mid` to the end of the slice.

When we try to compile the code in Listing 19-5, we'll get an error.

```
error[E0499]: cannot borrow `*slice` as mutable more than once at :
-->
|
6 |     (&mut slice[..mid],
|         ----- first mutable borrow occurs here
7 |     &mut slice[mid..])
|         ^^^^^ second mutable borrow occurs here
8 | }
| - first borrow ends here
```

Rust's borrow checker can't understand that we're borrowing different parts of the same slice. It knows that we're borrowing from the same slice twice. Borrowing different parts of the same slice is fundamentally okay because the two slices aren't overlapping, but Rust isn't smart enough to figure that out. When we know our code is okay, but Rust doesn't, it's time to reach for unsafe code.

Listing 19-6 shows how to use an `unsafe` block, a raw pointer, and some calls to the standard library to make the implementation of `split_at_mut` work.

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Listing 19-6: Using unsafe code in the implementation of the `split_at_mut` function

Recall from “The Slice Type” section in Chapter 4 that slices are a pointer to some memory and don't have their own length. We use the `len` method to get the length of a slice and the `as_mut_ptr` method to access the raw pointer of a slice. In this case, because we have a mutable reference to the slice, we can use `as_mut_ptr` to get a raw pointer. The `as_mut_ptr` method returns a raw pointer with the type `*mut i32`, which we've stored in the variable `ptr`.

We keep the assertion that the `mid` index is within the slice. Then we get to the `slice::from_raw_parts_mut` function. This function takes a raw pointer and a length, and it returns a slice that starts from `ptr` and is `mid` items long. Then we use the `offset` method on `ptr` with `mid` as an argument to get a raw pointer that starts at `ptr + mid`. Finally, we create a slice using that pointer and the remaining number of items after `mid` as the length.

The function `slice::from_raw_parts_mut` is unsafe because it takes a raw pointer as an argument. It's up to us to make sure that this pointer is valid. The `offset` method on raw pointers is also unsafe because it takes a raw pointer as an argument and makes sure that the offset location is also a valid pointer. Therefore, we had to put an `unsafe` block around the calls to `slice::from_raw_parts_mut` and `offset` so we could call them. By putting the `assert!` check inside the `unsafe` block, we can make sure that `mid` is less than or equal to `len`.

pointers used within the `unsafe` block will be valid pointers to data within the acceptable and appropriate use of `unsafe`.

Note that we don't need to mark the resulting `split_at_mut` function as `unsafe` from this function from safe Rust. We've created a safe abstraction to the unsafe implementation of the function that uses `unsafe` code in a safe way, because the pointers from the data this function has access to.

In contrast, the use of `slice::from_raw_parts_mut` in Listing 19-7 would likely be used. This code takes an arbitrary memory location and creates a slice of 10,

```
use std::slice;

let address = 0x012345usize;
let r = address as *mut i32;

let slice = unsafe {
    slice::from_raw_parts_mut(r, 10000)
};
```

Listing 19-7: Creating a slice from an arbitrary memory location

We don't own the memory at this arbitrary location, and there is no guarantee that the code contains valid `i32` values. Attempting to use `slice` as though it contained defined behavior is undefined behavior.

Using `extern` Functions to Call External Code

Sometimes, your Rust code might need to interact with code written in another language. Rust has a keyword, `extern`, that facilitates the creation and use of a *Foreign Function Interface*. An FFI is a way for a programming language to define functions and enable another programming language to call those functions.

Listing 19-8 demonstrates how to set up an integration with the `abs` function from the C standard library. Functions declared within `extern` blocks are always unsafe to call from Rust. The reason is that other languages don't enforce Rust's rules and guarantees, and so responsibility falls on the programmer to ensure safety.

Filename: `src/main.rs`

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Listing 19-8: Declaring and calling an `extern` function defined in another language

Within the `extern "C"` block, we list the names and signatures of external functions from the language we want to call. The `"C"` part defines which *application binary interface* (ABI) the function uses: the ABI defines how to call the function at the assembly level. Rust common and follows the C programming language's ABI.

Calling Rust Functions from Other Languages

We can also use `extern` to create an interface that allows other language functions. Instead of an `extern` block, we add the `extern` keyword and `as` just before the `fn` keyword. We also need to add a `#[no_mangle]` annotation to tell the compiler not to mangle the name of this function. *Mangling* is when a compiler gives a function a different name that contains more information about parts of the compilation process to consume but is less human readable. Different language compilers mangle names slightly differently, so for a Rust function that we want to be called from other languages, we must disable the Rust compiler's name mangling.

In the following example, we make the `call_from_c()` function accessible from C by compiling it to a shared library and linking from C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

This usage of `extern` does not require `unsafe`.

Accessing or Modifying a Mutable Static Variable

Until now, we've not talked about *global variables*, which Rust does support but only with Rust's ownership rules. If two threads are accessing the same mutable variable at the same time, it will cause a data race.

In Rust, global variables are called *static* variables. Listing 19-9 shows an example of defining and using a static variable with a string slice as a value.

Filename: `src/main.rs`

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

Listing 19-9: Defining and using an immutable static variable

Static variables are similar to constants, which we discussed in the “Different Types of Variables” section in Chapter 3. The names of static variables are in `SCREAMING_SNAKE_CASE` convention, and we *must* annotate the variable's type, which is `&'static str`. Static variables can only store references with the `'static` lifetime, which means they must live as long as the program. Rust figures out the lifetime; we don't need to annotate it explicitly. Accessing an immutable static variable is safe.

Constants and immutable static variables might seem similar, but a subtle difference between them is that static variables have a fixed address in memory. Using the value will always point to the same memory location. Constants, on the other hand, are allowed to duplicate their data whenever they are copied.

Another difference between constants and static variables is that static variables are *immutable*. Accessing and modifying mutable static variables is *unsafe*. Listing 19-10 shows how to access and modify a mutable static variable named `COUNTER`.

Filename: `src/main.rs`

```

static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}

```

Listing 19-10: Reading from or writing to a mutable static variable is unsafe

As with regular variables, we specify mutability using the `mut` keyword. Any writes from `COUNTER` must be within an `unsafe` block. This code compiles as we would expect because it's single threaded. Having multiple threads access result in data races.

With mutable data that is globally accessible, it's difficult to ensure there are why Rust considers mutable static variables to be unsafe. Where possible, it's concurrency techniques and thread-safe smart pointers we discussed in Chapter 18 checks that data accessed from different threads is done safely.

Implementing an Unsafe Trait

The final action that works only with `unsafe` is implementing an unsafe trait at least one of its methods has some invariant that the compiler can't verify. trait is `unsafe` by adding the `unsafe` keyword before `trait` and marking the trait as `unsafe` too, as shown in Listing 19-11.

```

unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}

```

Listing 19-11: Defining and implementing an unsafe trait

By using `unsafe impl`, we're promising that we'll uphold the invariants that

As an example, recall the `Sync` and `Send` marker traits we discussed in the with the `Sync` and `Send` Traits" section in Chapter 16: the compiler implements automatically if our types are composed entirely of `Send` and `Sync` types. If that contains a type that is not `Send` or `Sync`, such as raw pointers, and we as `Send` or `Sync`, we must use `unsafe`. Rust can't verify that our type upholds those checks manually and indicate as such with `unsafe`.

When to Use Unsafe Code

Using `unsafe` to take one of the four actions (superpowers) just discussed is frowned upon. But it is trickier to get `unsafe` code correct because the compiler can't check memory safety. When you have a reason to use `unsafe` code, you can do so with the `unsafe` annotation. This makes it easier to track down the source of problems if they occur.

Advanced Lifetimes

In Chapter 10 in the “Validating References with Lifetimes” section, you learned how to work with references with lifetime parameters to tell Rust how lifetimes of different references relate to each other. Every reference has a lifetime, but most of the time, Rust will let you elide the lifetime parameters. In this chapter, we’ll look at three advanced features of lifetimes that we haven’t covered yet:

- Lifetime subtyping: ensures that one lifetime outlives another lifetime
- Lifetime bounds: specifies a lifetime for a reference to a generic type
- Inference of trait object lifetimes: allows the compiler to infer trait object lifetimes when they need to be specified

Ensuring One Lifetime Outlives Another with Lifetime Subtyping

Lifetime subtyping specifies that one lifetime should outlive another lifetime. To demonstrate lifetime subtyping, imagine we want to write a parser. We’ll use a structure called `Context` to hold a reference to the string we’re parsing. We’ll write a parser that will parse this string into tokens of success or failure. The parser will need to borrow the `Context` to do the parsing. Listing 19-12 shows the code that implements this parser code, except the code doesn’t have the required lifetime annotations and won’t compile.

Filename: `src/lib.rs`

```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-12: Defining a parser without lifetime annotations

Compiling the code results in errors because Rust expects lifetime parameters for the `Parser` struct. The `context` field needs to have a lifetime parameter, and the reference to a `Context` in `Parser` needs to have a lifetime parameter.

For simplicity’s sake, the `parse` function returns `Result<(), &str>`. That is, it returns `Ok(())` on success and, on failure, will return the part of the string slice that contains the error message. A real implementation would provide more error information and would return `Err(())` when parsing succeeds. We won’t be discussing those details because they aren’t relevant to the lifetimes part of this example.

To keep this code simple, we won’t write any parsing logic. However, it’s very useful to understand what the parsing logic would do: it would handle invalid input by returning an error that contains a reference to the input that is invalid; this reference is what makes the code example interesting. Let’s pretend that the logic of our parser is that the input is invalid if the first byte is not a valid character boundary. Then this code might panic if the first byte is not a valid character boundary.

simplifying the example to focus on the lifetimes involved.

To get this code to compile, we need to fill in the lifetime parameters for the `Context` and the reference to the `Context` in `Parser`. The most straightforward way is to give them the same lifetime name everywhere, as shown in Listing 19-13. Recall from the “Struct Definitions” section in Chapter 10 that each of `struct Context<'a>`, `impl<'a> Parser<'a>` is declaring a new lifetime parameter. While their names happen to be the same, three lifetime parameters declared in this example aren’t related.

Filename: `src/lib.rs`

```
struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Listing 19-13: Annotating all references in `Context` and `Parser` with lifetime parameters

This code compiles just fine. It tells Rust that a `Parser` holds a reference to a string slice that lives as long as the `Parser` does, and that `Context` holds a string slice that also lives as long as the reference to it. Rust’s compiler error message stated that lifetime parameters were missing from references, and we’ve now added lifetime parameters.

Next, in Listing 19-14, we’ll add a function that takes an instance of `Context` and returns what `parse` returns. This code doesn’t quite work.

Filename: `src/lib.rs`

```
fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-14: An attempt to add a `parse_context` function that takes a `Context` and returns its result

We get two verbose errors when we try to compile the code with the additional function:

```

error[E0597]: borrowed value does not live long enough
--> src/lib.rs:14:5
|
14 |     Parser { context: &context }.parse()
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does not live long enough
15 |
| - temporary value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined
function body at 13:1...
--> src/lib.rs:13:1
|
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
| |_>

error[E0597]: `context` does not live long enough
--> src/lib.rs:14:24
|
14 |     Parser { context: &context }.parse()
|     ^^^^^^^ does not live long enough
15 |
| - borrowed value only lives until here
|
note: borrowed value must be valid for the anonymous lifetime #1 defined
function body at 13:1...
--> src/lib.rs:13:1
|
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
| |_>

```

These errors state that the `Parser` instance that is created and the `context` variable must both live until the end of the `parse_context` function. But they both need to live for the entire duration of the `parse` function.

In other words, `Parser` and `context` need to *outlive* the entire function and its body. The function starts as well as after it ends for all the references in this code to `Parser`. That means the `Parser` we're creating and the `context` parameter go out of scope at the end of the function because `parse_context` takes ownership of `context`.

To figure out why these errors occur, let's look at the definitions in Listing 19. Let's take a closer look at the references in the signature of the `parse` method:

```
fn parse(&self) -> Result<(), &str> {
```

Remember the elision rules? If we annotate the lifetimes of the references in the signature, it would be as follows:

```
fn parse<'a>(&'a self) -> Result<(), &'a str> {
```

That is, the error part of the return value of `parse` has a lifetime that is tied to the `Parser` instance (that of `&self` in the `parse` method signature). That makes the string slice references in the `Result` have the same lifetime. The lifetime of the `Parser` struct specifies that the lifetime of the reference to `Context` and the string slice that `Context` holds should be the same.

The problem is that the `parse_context` function returns the value returned by `parse`. The lifetime of the return value of `parse_context` is tied to the lifetime of the `Parser` instance created in the `parse_context` function. That means the `Parser` instance created in the `parse_context` function won't live past the end of the `parse_context` function.

temporary), and `context` will go out of scope at the end of the function (pa ownership of it).

Rust thinks we're trying to return a reference to a value that goes out of scop function, because we annotated all the lifetimes with the same lifetime para told Rust the lifetime of the string slice that `Context` holds is the same as th reference to `Context` that `Parser` holds.

The `parse_context` function can't see that within the `parse` function, the st outlive `Context` and `Parser` and that the reference `parse_context` returns slice, not to `Context` OR `Parser`.

By knowing what the implementation of `parse` does, we know that the only of `parse` is tied to the `Parser` instance is that it's referencing the `Parser` ir which is referencing the string slice. So, it's really the lifetime of the string sli needs to care about. We need a way to tell Rust that the string slice in `Conte` the `Context` in `Parser` have different lifetimes and that the return value of to the lifetime of the string slice in `Context`.

First, we'll try giving `Parser` and `Context` different lifetime parameters, as s. We'll use '`s`' and '`c`' as lifetime parameter names to clarify which lifetime g in `Context` and which goes with the reference to `context` in `Parser`. Note completely fix the problem, but it's a start. We'll look at why this fix isn't suffi compile.

Filename: `src/lib.rs`

```
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
    context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
    fn parse(&self) -> Result<(), &'s str> {
        Err(&self.context.0[1..])
    }
}

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

Listing 19-15: Specifying different lifetime parameters for the references to t Context

We've annotated the lifetimes of the references in all the same places that w Listing 19-13. But this time we used different parameters depending on whe with the string slice or with `context`. We've also added an annotation to the return value of `parse` to indicate that it goes with the lifetime of the string s

When we try to compile now, we get the following error:

```

error[E0491]: in type `&'c Context<'s>`, reference has a longer li-
it references
--> src/lib.rs:4:5
  |
4 |     context: &'c Context<'s>,
  |     ^^^^^^^^^^^^^^^^^^^^^^
  |
  note: the pointer is valid for the lifetime 'c as defined on the s
--> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_ ^
  note: but the referenced data is only valid for the lifetime 's as
struct at 3:1
--> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_ ^

```

Rust doesn't know of any relationship between '`c`' and '`s`'. To be valid, the `Context` with lifetime '`s`' needs to be constrained to guarantee that it lives reference with lifetime '`c`'. If '`s`' is not longer than '`c`', the reference to `co` valid.

Now we get to the point of this section: the Rust feature *lifetime subtyping* sp parameter lives at least as long as another one. In the angle brackets where parameters, we can declare a lifetime '`a`' as usual and declare a lifetime '`b`' long as '`a`' by declaring '`b`' using the syntax '`b: 'a`'.

In our definition of `Parser`, to say that '`s`' (the lifetime of the string slice) is least as long as '`c`' (the lifetime of the reference to `Context`), we change thi look like this:

Filename: `src/lib.rs`

```

struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}

```

Now the reference to `Context` in the `Parser` and the reference to the string slice have different lifetimes; we've ensured that the lifetime of the string slice is longer than the reference to the `context`.

That was a very long-winded example, but as we mentioned at the start of this chapter, lifetimes and advanced features are very specific. You won't often need the syntax we described here, but in such situations, you'll know how to refer to something and give it the right lifetime.

Lifetime Bounds on References to Generic Types

In the "Trait Bounds" section in Chapter 10, we discussed using trait bounds to constrain generic types. We can also add lifetime parameters as constraints on generic types; these are called lifetime bounds. Lifetime bounds help Rust verify that references in generic types won't outlive the objects they're referencing.

As an example, consider a type that is a wrapper over references. Recall the `Box` type from Chapter 10:

the “`RefCell<T>` and the Interior Mutability Pattern” section in Chapter 15: `borrow_mut` methods return the types `Ref` and `RefMut`, respectively. These are references that keep track of the borrowing rules at runtime. The definition shown in Listing 19-16, without lifetime bounds for now.

Filename: `src/lib.rs`

```
struct Ref<'a, T>(&'a T);
```

Listing 19-16: Defining a struct to wrap a reference to a generic type, without lifetime bounds

Without explicitly constraining the lifetime `'a` in relation to the generic parameter `T`, the code fails to compile because it doesn’t know how long the generic type `T` will live:

```
error[E0309]: the parameter type `T` may not live long enough
--> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |          ^^^^^^
  |
  = help: consider adding an explicit lifetime bound `T: 'a`...
note: ...so that the reference type `&'a T` does not outlive the data it points at
--> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |          ^^^^^^
```

Because `T` can be any type, `T` could be a reference or a type that holds one or more references, each of which could have their own lifetimes. Rust can’t be sure `T` will live as long as its references.

Fortunately, the error provides helpful advice on how to specify the lifetime bounds:

```
consider adding an explicit lifetime bound `T: 'a` so that the reference type `&'a T` does not outlive the data it points at
```

Listing 19-17 shows how to apply this advice by specifying the lifetime bounds for the generic type `T`.

```
struct Ref<'a, T: 'a>(&'a T);
```

Listing 19-17: Adding lifetime bounds on `T` to specify that any references in `T` must live at least as long as `'a`.

This code now compiles because the `T: 'a` syntax specifies that `T` can be a type that contains references, and if `T` contains references, the references must live at least as long as `'a`.

We could solve this problem in a different way, as shown in the definition of `StaticRef` in Listing 19-18, by adding the `'static` lifetime bound on `T`. This means if `T` contains references, they must have the `'static` lifetime.

```
struct StaticRef<T: 'static>(&'static T);
```

Listing 19-18: Adding a `'static` lifetime bound to `T` to constrain `T` to types that contain no references or only `'static` references.

Because `'static` means the reference must live as long as the entire program (there are no references that outlive the entire program), this means `T` must contain no references (because all references in `T` must live as long as the entire program, and there are no references). For the borrow checker concerned about references living longer than the entire program, this means `T` must be `'static`.

no real distinction between a type that has no references and a type that has forever: both are the same for determining whether or not a reference has a what it refers to.

Inference of Trait Object Lifetimes

In Chapter 17 in the “Using Trait Objects that Allow for Values of Different Types” section, we discussed trait objects, consisting of a trait behind a reference, that allow us to dispatch. We haven’t yet discussed what happens if the type implementing the trait has a lifetime of its own. Consider Listing 19-19 where we have a trait `Red` and a `Ball` struct holds a reference (and thus has a lifetime parameter) and also implements the `Red` trait. We want to use an instance of `Ball` as the trait object `Box<dyn Red>`.

Filename: `src/main.rs`

```
trait Red { }

struct Ball<'a> {
    diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
    let num = 5;

    let obj = Box::new(Ball { diameter: &num }) as Box<dyn Red>;
}
```

Listing 19-19: Using a type that has a lifetime parameter with a trait object

This code compiles without any errors, even though we haven’t explicitly announced the lifetime parameter `'a` involved in `obj`. This code works because there are rules for working with lifetimes:

- The default lifetime of a trait object is `'static`.
- With `&'a Trait` or `&'a mut Trait`, the default lifetime of the trait object is `'a`.
- With a single `T: 'a` clause, the default lifetime of the trait object is `'a`.
- With multiple clauses like `T: 'a`, there is no default lifetime; we must be explicit.

When we must be explicit, we can add a lifetime bound on a trait object like `Box<dyn Red + 'static>` or `Box<dyn Red + 'a>`, depending on whether the lifetime applies for the entire program or not. As with the other bounds, the syntax adding a lifetime to a trait object means that any implementor of the `Red` trait that has references inside the type must respect the lifetime specified in the trait object bounds as those references.

Next, let’s look at some other advanced features that manage traits.

Advanced Traits

We first covered traits in the “Traits: Defining Shared Behavior” section of Chapter 17. While we covered the basics of lifetimes, we didn’t discuss the more advanced details. Now that you know more about traits, let’s get into the nitty-gritty.

Specifying Placeholder Types in Trait Definitions with Associated Types

Associated types connect a type placeholder with a trait such that the trait means these placeholder types in their signatures. The implementor of a trait will specify to be used in this type's place for the particular implementation. That way, we can use some types without needing to know exactly what those types are until implemented.

We've described most of the advanced features in this chapter as being rare because types are somewhere in the middle: they're used more rarely than features like closures in the book but more commonly than many of the other features discussed in this chapter.

One example of a trait with an associated type is the `Iterator` trait that the standard library provides. The associated type is named `Item` and stands in for the type of the value being iterated over. In "The `Iterator` Trait" section of Chapter 13, we mentioned that the definition of the `Iterator` trait is in Listing 19-20.

```
pub trait Iterator {
    type Item;

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

Listing 19-20: The definition of the `Iterator` trait that has an associated type

The type `Item` is a placeholder type, and the `next` method's definition shows that it returns values of type `Option<Self::Item>`. Implementors of the `Iterator` trait will define the type for `Item`, and the `next` method will return an `Option` containing a value of that type.

Associated types might seem like a similar concept to generics, in that the language lets you define a function without specifying what types it can handle. So why use associated types?

Let's examine the difference between the two concepts with an example from Listing 19-21. It implements the `Iterator` trait on the `Counter` struct. In Listing 13-21, we saw that the `Counter` type was `u32`:

Filename: `src/lib.rs`

```
impl Iterator for Counter {
    type Item = u32;

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

This syntax seems comparable to that of generics. So why not just define the `Counter` type using generics, as shown in Listing 19-21?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Listing 19-21: A hypothetical definition of the `Iterator` trait using generics

The difference is that when using generics, as in Listing 19-21, we must annotate the `Counter` type with the generic parameter `T` to provide the implementation; because we can also implement `Iterator<String>` for `Counter`, we could have multiple implementations of `Iterator` for `Counter`. In other words, with a generic parameter, it can be implemented for a type multiple times, changing the generic type parameters each time. When we use the `next` method on `Counter`, we must provide type annotations to indicate which implementation of `Iterator` to use:

With associated types, we don't need to annotate types because we can't implement multiple times. In Listing 19-20 with the definition that uses associated types what the type of `Item` will be once, because there can only be one `impl Iterator`. We don't have to specify that we want an iterator of `u32` values everywhere that `Counter`.

Default Generic Type Parameters and Operator Overloading

When we use generic type parameters, we can specify a default concrete type. This eliminates the need for implementors of the trait to specify a concrete type. The syntax for specifying a default type for a generic type is `<PlaceholderType=ConcreteType>` when declaring the generic type.

A great example of a situation where this technique is useful is with operator *overloading* is customizing the behavior of an operator (such as `+`) in particular.

Rust doesn't allow you to create your own operators or overload arbitrary operators. Instead, you can overload the operations and corresponding traits listed in `std::ops` by implementing the trait associated with the operator. For example, in Listing 19-22 we overload the `+` operator for `Point` instances together. We do this by implementing the `Add` trait on a `Point`.

Filename: `src/main.rs`

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

Listing 19-22: Implementing the `Add` trait to overload the `+` operator for `Point`

The `add` method adds the `x` values of two `Point` instances and the `y` values of two `Point` instances to create a new `Point`. The `Add` trait has an associated type named `Output`, which determines the type returned from the `add` method.

The default generic type in this code is within the `Add` trait. Here is its definition:

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

This code should look generally familiar: a trait with one method and an associated part is `RHS=Self`: this syntax is called *default type parameters*. The `RHS` generic parameter (short for “right hand side”) defines the type of the `rhs` parameter in the `add` method. We can specify a concrete type for `RHS` when we implement the `Add` trait, the type `Self`, which will be the type we’re implementing `Add` on.

When we implemented `Add` for `Point`, we used the default for `RHS` because `Point` instances implement the `Copy` trait. Let’s look at an example of implementing the `Add` trait where we can customize the `RHS` type rather than using the default.

We have two structs, `Millimeters` and `Meters`, holding values in different units. We want to convert values in millimeters to values in meters and have the implementation of `Add` work correctly. We can implement `Add` for `Millimeters` with `Meters` as the `RHS`. Listing 19-23 shows the code for Listing 19-23.

Filename: `src/lib.rs`

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Listing 19-23: Implementing the `Add` trait on `Millimeters` to add `Millimeters` to `Meters`.

To add `Millimeters` and `Meters`, we specify `impl Add<Meters>` to set the `RHS` parameter instead of using the default of `Self`.

You’ll use default type parameters in two main ways:

- To extend a type without breaking existing code
- To allow customization in specific cases most users won’t need

The standard library’s `Add` trait is an example of the second purpose: usually you extend a type by adding methods to it, but the `Add` trait provides the ability to customize beyond that. Using `Self` as the `RHS` parameter in the `Add` trait definition means you don’t have to specify the exact type every time. In other words, a bit of implementation boilerplate isn’t needed, making the trait more general.

The first purpose is similar to the second but in reverse: if you want to add a trait to an existing type, you can give it a default to allow extension of the functionality without breaking the existing implementation code.

Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name

Nothing in Rust prevents a trait from having a method with the same name as another trait’s method, nor does Rust prevent you from implementing both traits on one type. This means you can implement a method directly on the type with the same name as methods from multiple traits.

When calling methods with the same name, you’ll need to tell Rust which one you want to call. Consider the code in Listing 19-24 where we’ve defined two traits, `Pilot` and `Captain`.

have a method called `fly`. We then implement both traits on a type `Human` method named `fly` implemented on it. Each `fly` method does something

Filename: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

Listing 19-24: Two traits are defined to have a `fly` method and are implemented on the `Human` type, and a `fly` method is implemented on `Human` directly.

When we call `fly` on an instance of `Human`, the compiler defaults to calling the `fly` method directly implemented on the type, as shown in Listing 19-25.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    person.fly();
}
```

Listing 19-25: Calling `fly` on an instance of `Human`

Running this code will print `*waving arms furiously*`, showing that Rust calls the `fly` method implemented on `Human` directly.

To call the `fly` methods from either the `Pilot` trait or the `Wizard` trait, we need explicit syntax to specify which `fly` method we mean. Listing 19-26 demonstrates this.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-26: Specifying which trait's `fly` method we want to call

Specifying the trait name before the method name clarifies to Rust which implementation of the `fly` method we want to call. We could also write `Human::fly(&person)`, which is equivalent to what we used in Listing 19-26, but this is a bit longer to write if we don't need to.

Running this code prints the following:

```
This is your captain speaking.  
Up!  
*waving arms furiously*
```

Because the `fly` method takes a `self` parameter, if we had two *types* that implement the *trait*, Rust could figure out which implementation of a trait to use based on the type of the `self` parameter.

However, associated functions that are part of traits don't have a `self` parameter, so if two types implement the same trait, Rust can't figure out which type you intended to use. For example, the `Animal` trait in Listing 19-27 has the `baby_name` associated function, the implementation of `Animal` for the struct `Dog`, and the associated function `baby_name` defined on `Dog` directly.

Filename: `src/main.rs`

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

Listing 19-27: A trait with an associated function and a type with an associated function that also implements the trait

This code is for an animal shelter that wants to name all puppies Spot, which is the `baby_name` associated function that is defined on `Dog`. The `Dog` type also implements the `Animal` trait, which describes characteristics that all animals have. Baby dogs are puppies, which is expressed in the implementation of the `Animal` trait on `Dog` in the `impl Animal for Dog` block. The `baby_name` function is associated with the `Animal` trait.

In `main`, we call the `Dog::baby_name` function, which calls the associated function directly. This code prints the following:

```
A baby dog is called a Spot
```

This output isn't what we wanted. We want to call the `baby_name` function from the `Animal` trait that we implemented on `Dog` so the code prints `A baby dog is called a puppy`. The technique of specifying the trait name that we used in Listing 19-26 doesn't help here.

main to the code in Listing 19-28, we'll get a compilation error.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

Listing 19-28: Attempting to call the `baby_name` function from the `Animal` trait without knowing which implementation to use

Because `Animal::baby_name` is an associated function rather than a method with a `self` parameter, Rust can't figure out which implementation of `Animal::baby_name` to use. We'll get this compiler error:

```
error[E0283]: type annotations required: cannot resolve `_: Animal`
--> src/main.rs:20:43
   |
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                     ^^^^^^^^^^^^^^^^^^
   |
   = note: required by `Animal::baby_name`
```

To disambiguate and tell Rust that we want to use the implementation of `Animal::baby_name`, we need to use fully qualified syntax. Listing 19-29 demonstrates how to use fully qualified syntax.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Listing 19-29: Using fully qualified syntax to specify that we want to call the `baby_name` method from the `Animal` trait as implemented on `Dog`

We're providing Rust with a type annotation within the angle brackets, which tells Rust to call the `baby_name` method from the `Animal` trait as implemented on `Dog`. This tells Rust to treat the `Dog` type as an `Animal` for this function call. This code will now print:

A baby dog is called a puppy

In general, fully qualified syntax is defined as follows:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

For associated functions, there would not be a `receiver`: there would only be arguments. You could use fully qualified syntax everywhere that you call functions. However, you're allowed to omit any part of this syntax that Rust can figure out from context or information in the program. You only need to use this more verbose syntax when you have multiple implementations that use the same name and Rust needs help to identify which implementation you want to call.

Using Supertraits to Require One Trait's Functionality Within Another

Sometimes, you might need one trait to use another trait's functionality. In this case, you rely on the dependent trait's also being implemented. The trait you rely on is the trait you're implementing.

For example, let's say we want to make an `OutlinePrint` trait with an `outline` method that prints its argument with a border:

will print a value framed in asterisks. That is, given a `Point` struct that implements `outline_print`, the result in `(x, y)`, when we call `outline_print` on a `Point` instance that has `outline_print` implemented, it should print the following:

```
*****
*      *
* (1, 3) *
*      *
*****
```

In the implementation of `outline_print`, we want to use the `Display` trait. Therefore, we need to specify that the `outlinePrint` trait will work only for types that implement `Display` and provide the functionality that `OutlinePrint` needs. We do this by specifying `OutlinePrint: Display`. This technique is similar to how we bound traits to specific types in the previous section. It's also bound to the trait. Listing 19-30 shows an implementation of the `OutlinePrint` trait.

Filename: `src/main.rs`

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}{}", "*".repeat(len + 4));
    }
}
```

Listing 19-30: Implementing the `OutlinePrint` trait that requires the `Display` trait

Because we've specified that `OutlinePrint` requires the `Display` trait, we can't implement `outline_print` directly on `Point`. Instead, we can implement the `Display` trait on `Point` and then implement `outline_print` on `Point` using the `Display` trait. If we try to implement `outline_print` directly on `Point`, we'll get an error saying that no method named `to_string` was found for the type `&Point`.

Let's see what happens when we try to implement `OutlinePrint` on a type that doesn't implement `Display`, such as the `Point` struct:

Filename: `src/main.rs`

```
struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

We get an error saying that `Display` is required but not implemented:

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
   --> src/main.rs:20:6
    |
20 | impl OutlinePrint for Point {}  
|     ^^^^^^^^^^^^^ `Point` cannot be formatted with the default  
try using `{:?}` instead if you are using a format string  
|  
= help: the trait `std::fmt::Display` is not implemented for `Point`
```

To fix this, we implement `Display` on `Point` and satisfy the constraint that it like so:

Filename: `src/main.rs`

```
use std::fmt;  
  
impl fmt::Display for Point {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        write!(f, "({}, {})", self.x, self.y)  
    }  
}
```

Then implementing the `OutlinePrint` trait on `Point` will compile successfully. We can now call `outline_print` on a `Point` instance to display it within an outline of asterisks.

Using the Newtype Pattern to Implement External Traits on Internal Types

In Chapter 10 in the “Implementing a Trait on a Type” section, we mentioned that we’re allowed to implement a trait on a type as long as either the trait or our crate. It’s possible to get around this restriction using the *newtype pattern*: creating a new type in a tuple struct. (We covered tuple structs in the “Using Named Fields to Create Different Types” section of Chapter 5.) The tuple struct will be a thin wrapper around the type we want to implement a trait for. The type will be local to our crate, and we can implement the trait on the wrapper. *Newtype* is a pattern from the Haskell programming language. There is no runtime performance cost from using this pattern, and the wrapper type is elided at compile time.

As an example, let’s say we want to implement `Display` on `Vec<T>`, which is not possible because the `Display` trait and the `Vec<T>` type are completely separate. We can make a `Wrapper` struct that holds an instance of `Vec<T>`; then we can implement `Display` on `Wrapper` and use the `Vec<T>` value, as shown in Listing 19-31.

Filename: `src/main.rs`

```
use std::fmt;  
  
struct Wrapper(Vec<String>);  
  
impl fmt::Display for Wrapper {  
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
        write!(f, "[{}]", self.0.join(", "))  
    }  
}  
  
fn main() {  
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);  
    println!("w = {}", w);  
}
```

Listing 19-31: Creating a `Wrapper` type around `Vec<String>` to implement `Display` on it

The implementation of `Display` uses `self.0` to access the inner `Vec<T>`, tuple struct and `vec<T>` is the item at index 0 in the tuple. Then we can use `Display` type on `Wrapper`.

The downside of using this technique is that `Wrapper` is a new type, so it does not have access to the methods of the value it's holding. We would have to implement all the methods of `Vec<T>` and `Wrapper` such that the methods delegate to `self.0`, which would allow us to use them on `Wrapper` like a `Vec<T>`. If we wanted the new type to have every method of the inner type, we could implement the `Deref` trait (discussed in Chapter 15 in the “Treating Smart Pointers like Reference Types” section) on the `Wrapper` to return the inner type. However, if we want the `Wrapper` type to have all the methods of the inner type—for example, `Display`—we would have to implement just the methods we care about.

Now you know how the newtype pattern is used in relation to traits; it's also useful when traits are not involved. Let's switch focus and look at some advanced uses of Rust's type system.

Advanced Types

The Rust type system has some features that we've mentioned in this book but haven't discussed. We'll start by discussing newtypes in general as we examine why they're useful. Then we'll move on to type aliases, a feature similar to newtypes but with different semantics. We'll also discuss the `!` type and dynamically sized types.

Note: The next section assumes you've read the earlier section “The Newtype Pattern for Type Safety and Abstraction”.

Using the Newtype Pattern for Type Safety and Abstraction

The newtype pattern is useful for tasks beyond those we've discussed so far, such as enforcing that values are never confused and indicating the units of a value. For example, in Listing 19-23, we used `Millimeters` to wrap `u32` values in a newtype. If we wrote a function with a parameter of type `Millimeters`, it couldn't compile a program that accidentally tried to call that function with a plain `u32`.

Another use of the newtype pattern is in abstracting away some implementation details. For example, if we had a `Person` type that wrapped a `HashMap<i32, String>` that stores a person's ID associated with their name, we could wrap the `Person` type in a newtype. The newtype would only interact with the public API we provide, such as a method to add a person to the `People` collection; that code wouldn't need to know that we assign an ID internally. The newtype pattern is a lightweight way to achieve encapsulation without hiding implementation details, which we discussed in the “Encapsulation that Hides Implementation Details” section of Chapter 17.

Creating Type Synonyms with Type Aliases

Along with the newtype pattern, Rust provides the ability to declare a *type alias*.

type another name. For this we use the `type` keyword. For example, we can create an alias `Kilometers` to `i32` like so:

```
type Kilometers = i32;
```

Now, the alias `Kilometers` is a *synonym* for `i32`; unlike the `Millimeters` created in Listing 19-23, `Kilometers` is not a separate, new type. Values that have the `Kilometers` type will be treated the same as values of type `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Because `Kilometers` and `i32` are the same type, we can add values of both `Kilometers` values to functions that take `i32` parameters. However, using `Kilometers` gets the type checking benefits that we get from the newtype pattern discussed.

The main use case for type synonyms is to reduce repetition. For example, we can write code like this:

```
Box<dyn Fn() + Send + 'static>
```

Writing this lengthy type in function signatures and as type annotations all over a function is tiresome and error prone. Imagine having a project full of code like that in Listing 19-32:

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --snip--
}
```

Listing 19-32: Using a long type in many places

A type alias makes this code more manageable by reducing the repetition. We introduced an alias named `Thunk` for the verbose type and can replace all uses of the shorter alias `Thunk`.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Listing 19-33: Introducing a type alias `Thunk` to reduce repetition

This code is much easier to read and write! Choosing a meaningful name for communicate your intent as well (*thunk* is a word for code to be evaluated at appropriate name for a closure that gets stored).

Type aliases are also commonly used with the `Result<T, E>` type for reduce the `std::io` module in the standard library. I/O operations often return a `Result` situations when operations fail to work. This library has a `std::io::Error` type for possible I/O errors. Many of the functions in `std::io` will be returning `Result<T, std::io::Error>`, such as these functions in the `Write` trait:

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

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

The `Result<..., Error>` is repeated a lot. As such, `std::io` has this type definition:

```
type Result<T> = Result<T, std::io::Error>;
```

Because this declaration is in the `std::io` module, we can use the fully qualified type `std::io::Result<T>`—that is, a `Result<T, E>` with the `E` filled in as `std::io::Error`. The trait function signatures end up looking like this:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

The type alias helps in two ways: it makes code easier to write *and* it gives us a single place to change behavior across all of `std::io`. Because it's an alias, it's just another `Result<T, E>`, which means we can use any methods that work on `Result<T, E>` with it, as well as special syntax like `match`.

The Never Type that Never Returns

Rust has a special type named `!` that's known in type theory lingo as the *empty type* or *unit type*. It represents no values. We prefer to call it the *never type* because it stands in the place of a value that a function will never return. Here is an example:

```
fn bar() -> ! {
    // --snip--
}
```

This code is read as “the function `bar` returns never.” Functions that return `!` are called *diverging functions*. We can't create values of the type `!` so `bar` can never produce a value.

But what use is a type you can never create values for? Recall the code from Listing 19-33, and let's reproduce part of it here in Listing 19-34.

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

Listing 19-34: A `match` with an arm that ends in `continue`

At the time, we skipped over some details in this code. In Chapter 6 in “The `!` Operator” section, we discussed that `match` arms must all return the same type. The following code doesn’t work:

```
let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => "hello",
}
```

The type of `guess` in this code would have to be an integer *and* a string, and `guess` can have only one type. So what does `continue` return? How were we able to end from one arm and have another arm that ends with `continue` in Listing 19-34?

As you might have guessed, `continue` has a `!` value. That is, when Rust considers `guess`, it looks at both match arms, the former with a value of `u32` and the latter with a value of `!`. Because ! can never have a value, Rust decides that the type of guess is !.`

The formal way of describing this behavior is that expressions of type `!` can have any other type. We’re allowed to end this `match` arm with `continue` because `continue` is a value; instead, it moves control back to the top of the loop, so in the `Err` case, `continue` returns the value to `guess`.

The `never` type is useful with the `panic!` macro as well. Remember the `unwrap` method on `Option<T>` values to produce a value or panic? Here is its definition:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value")
        }
    }
}
```

In this code, the same thing happens as in the `match` in Listing 19-34: Rust sees that the type `T` and `panic!` has the type `!`, so the result of the overall `match` expression is `!`. This works because `panic!` doesn’t produce a value; it ends the program. In the `None` case, `unwrap` is returning a value from `unwrap`, so this code is valid.

One final expression that has the type `!` is a `loop`:

```
print!("forever ");
loop {
    print!("and ever ");
}
```

Here, the loop never ends, so `!` is the value of the expression. However, this code included a `break`, because the loop would terminate when it got to the `break` statement.

Dynamically Sized Types and the `Sized` Trait

Due to Rust's need to know certain details, such as how much space to allocate for a particular type, there is a corner of its type system that can be confusing: the *sized types*. Sometimes referred to as *DSTs* or *unsized types*, these types let us know whose size we can know only at runtime.

Let's dig into the details of a dynamically sized type called `str`, which we've seen in the book. That's right, not `&str`, but `str` on its own, is a DST. We can't know its size until runtime, meaning we can't create a variable of type `str`, nor can we take a reference to it. Consider the following code, which does not work:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust needs to know how much memory to allocate for any value of a particular type. Every value of a type must use the same amount of memory. If Rust allowed us to write `str` values, then `s1` would need 12 bytes of storage and `s2` would need 15. This is why it's not possible to create a dynamically sized type.

So what do we do? In this case, you already know the answer: we make the type `&str` rather than a `str`. Recall that in the "String Slices" section of Chapter 4, we learned that `&str` is a pointer to a string, and its structure stores the starting position and the length of the slice.

So although a `&T` is a single value that stores the memory address of where `T` is stored, `&str` is two values: the address of the `str` and its length. As such, we can know the size of `&str` at compile time: it's twice the length of a `usize`. That is, we always know the size of `&str`, no matter how long the string it refers to is. In general, this is the way in which dynamically sized types are used in Rust: they have an extra bit of metadata that stores the size of the value. The golden rule of dynamically sized types is that we must always put values behind a pointer of some kind.

We can combine `str` with all kinds of pointers: for example, `Box<str>` or `Rc<str>`. We've seen this before but with a different dynamically sized type: traits. Every trait is a type we can refer to by using the name of the trait. In Chapter 17 in the "Using Traits to Allow for Values of Different Types" section, we mentioned that to use traits with dynamically sized types, we put them behind a pointer, such as `&dyn Trait` or `Box<dyn Trait>` (`Rc<dyn Trait>` too).

To work with DSTs, Rust has a particular trait called the `sized` trait to determine the size of a type. The size of a type is known at compile time. This trait is automatically implemented for every type that has a known size. The size of a type is known at compile time. In addition, Rust implicitly adds a bound on `sized` to function definitions. That is, a generic function definition like this:

```
fn generic<T>(t: T) {
    // --snip--
}
```

is actually treated as though we had written this:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

By default, generic functions will work only on types that have a known size. However, you can use the following special syntax to relax this restriction:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

A trait bound on `?Sized` is the opposite of a trait bound on `Sized`: we would not be `Sized`." This syntax is only available for `Sized`, not any other traits.

Also note that we switched the type of the `t` parameter from `T` to `&T`. Because `Sized`, we need to use it behind some kind of pointer. In this case, we've

Next, we'll talk about functions and closures!

Advanced Functions and Closures

Finally, we'll explore some advanced features related to functions and closures: function pointers and returning closures.

Function Pointers

We've talked about how to pass closures to functions; you can also pass regular functions! This technique is useful when you want to pass a function you've defined than defining a new closure. Doing this with function pointers will allow you to pass arguments to other functions. Functions coerce to the type `fn` (with a lower case `f`) confused with the `Fn` closure trait. The `fn` type is called a *function pointer*. Notice that a parameter is a function pointer is similar to that of closures, as shown

Filename: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

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

Listing 19-35: Using the `fn` type to accept a function pointer as an argument

This code prints `The answer is: 12`. We specify that the parameter `f` in `do_twice` takes one parameter of type `i32` and returns an `i32`. We can then call `f` in `main`. In `main`, we can pass the function name `add_one` as the first argument to `do_twice`.

Unlike closures, `fn` is a type rather than a trait, so we specify `fn` as the parameter type rather than declaring a generic type parameter with one of the `Fn` traits as we did with closures.

Function pointers implement all three of the closure traits (`Fn`, `FnMut`, and `FnOnce`), so you can always pass a function pointer as an argument for a function that expects a closure. You can also write functions using a generic type and one of the closure traits so your functions accept both function pointers and closures.

An example of where you would want to only accept `fn` and not closures is

external code that doesn't have closures: C functions can accept functions as arguments, but C doesn't have closures.

As an example of where you could use either a closure defined inline or a named function as the argument at a use of `map`. To use the `map` function to turn a vector of numbers into a vector of strings, we could use a closure, like this:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

Or we could name a function as the argument to `map` instead of the closure

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

Note that we must use the fully qualified syntax that we talked about earlier in the section because there are multiple functions available named `to_string`. However, we're using the `ToString` trait's `to_string` function defined in the `ToString` trait, which the standard library provides for any type that implements `Display`.

Some people prefer this style, and some people prefer to use closures. They both work, so use whichever style is clearer to you.

Returning Closures

Closures are represented by traits, which means you can't return closures directly. If you want to return a closure where you might want to return a trait, you can instead use the concrete type as the return value of the function. But you can't do that with closures because they're not a concrete type that is returnable; you're not allowed to use the function pointer type, for example.

The following code tries to return a closure directly, but it won't compile:

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

The compiler error is as follows:

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static: std::marker::Sized` is not satisfied
-->
|
1 | fn returns_closure() -> Fn(i32) -> i32 {
|           ^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
|
= help: the trait `std::marker::Sized` is not implemented for `std::ops::Fn(i32) -> i32 + 'static'
= note: the return type of a function must have a statically known size
```

The error references the `Sized` trait again! Rust doesn't know how much space to allocate for the closure.

the closure. We saw a solution to this problem earlier. We can use a trait obj

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

This code will compile just fine. For more about trait objects, refer to the “Us Allow for Values of Different Types” section in Chapter 17.

Summary

Whew! Now you have some features of Rust in your toolbox that you won’t know they’re available in very particular circumstances. We’ve introduced several concepts that when you encounter them in error message suggestions or in other places, you can recognize these concepts and syntax. Use this chapter as a reference to get started.

Next, we’ll put everything we’ve discussed throughout the book into practice by building a final project!

Final Project: Building a Multithreaded Web Server

It’s been a long journey, but we’ve reached the end of the book. In this chapter, we’ll build a web server from scratch, piece by piece, to demonstrate some of the concepts we covered in the first half of the book. We’ll also recap some earlier lessons.

For our final project, we’ll make a web server that says “hello” and looks like this:



Hello!

Hi from Rust

Figure 20-1: Our final shared project

Here is the plan to build the web server:

1. Learn a bit about TCP and HTTP.
2. Listen for TCP connections on a socket.
3. Parse a small number of HTTP requests.
4. Create a proper HTTP response.
5. Improve the throughput of our server with a thread pool.

But before we get started, we should mention one detail: the method we'll use to build a web server with Rust. A number of production-ready crates are available at <https://crates.io/> that provide more complete web server and thread pool implementations.

However, our intention in this chapter is to help you learn, not to take the easy way out. As Rust is a systems programming language, we can choose the level of abstraction and can go to a lower level than is possible or practical in other languages. You will have to build the web server and thread pool manually so you can learn the general ideas and techniques that the production-ready crates you might use in the future.

Building a Single-Threaded Web Server

We'll start by getting a single-threaded web server working. Before we begin, let's take a brief overview of the protocols involved in building web servers. The details of the protocols are beyond the scope of this book, but a brief overview will give you the information you need.

The two main protocols involved in web servers are the *Hypertext Transfer Protocol (HTTP)* and the *Transmission Control Protocol (TCP)*. Both protocols are *request-response* protocols. A *client* initiates requests and a *server* listens to the requests and provides a response. The contents of those requests and responses are defined by the protocols.

TCP is the lower-level protocol that describes the details of how information is transferred between two hosts. It doesn't specify what that information is. HTTP builds on top of TCP to define the contents of the requests and responses. It's technically possible to use HTTP over UDP, but in the vast majority of cases, HTTP sends its data over TCP. We'll work with TCP in this chapter, and HTTP requests and responses.

Listening to the TCP Connection

Our web server needs to listen to a TCP connection, so that's the first part we'll implement. The standard library offers a `std::net` module that lets us do this. Let's make a simple application:

```
$ cargo new hello
     Created binary (application) `hello` project
$ cd hello
```

Now enter the code in Listing 20-1 in `src/main.rs` to start. This code will listen on port 7878 for incoming TCP streams. When it gets an incoming stream, it prints "Connection established!".

Filename: `src/main.rs`

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("Connection established!");
    }
}
```

Listing 20-1: Listening for incoming streams and printing a message when we receive one

Using `TcpListener`, we can listen for TCP connections at the address `127.0` address, the section before the colon is an IP address representing your con on every computer and doesn't represent the authors' computer specifically We've chosen this port for two reasons: HTTP is normally accepted on this p typed on a telephone.

The `bind` function in this scenario works like the `new` function in that it will `TcpListener` instance. The reason the function is called `bind` is that in net port to listen to is known as "binding to a port."

The `bind` function returns a `Result<T, E>`, which indicates that binding mi connecting to port 80 requires administrator privileges (nonadministrators c higher than 1024), so if we tried to connect to port 80 without being an adm wouldn't work. As another example, binding wouldn't work if we ran two insl and so had two programs listening to the same port. Because we're writing a learning purposes, we won't worry about handling these kinds of errors; inst stop the program if errors happen.

The `incoming` method on `TcpListener` returns an iterator that gives us a si (more specifically, streams of type `TcpStream`). A single `stream` represents a between the client and the server. A `connection` is the name for the full requ process in which a client connects to the server, the server generates a resp closes the connection. As such, `TcpStream` will read from itself to see what t allow us to write our response to the stream. Overall, this `for` loop will proc turn and produce a series of streams for us to handle.

For now, our handling of the stream consists of calling `unwrap` to terminate stream has any errors; if there aren't any errors, the program prints a messa functionality for the success case in the next listing. The reason we might rec `incoming` method when a client connects to the server is that we're not actu connections. Instead, we're iterating over `connection attempts`. The connectio successful for a number of reasons, many of them operating system specific operating systems have a limit to the number of simultaneous open connect new connection attempts beyond that number will produce an error until so connections are closed.

Let's try running this code! Invoke `cargo run` in the terminal and then load browser. The browser should show an error message like "Connection reset, isn't currently sending back any data. But when you look at your terminal, yo messages that were printed when the browser connected to the server!

```
Running `target/debug/hello`  
Connection established!  
Connection established!  
Connection established!
```

Sometimes, you'll see multiple messages printed for one browser request; th the browser is making a request for the page as well as a request for other r `favicon.ico` icon that appears in the browser tab.

It could also be that the browser is trying to connect to the server multiple ti isn't responding with any data. When `stream` goes out of scope and is drop| loop, the connection is closed as part of the `drop` implementation. Browser closed connections by retrying, because the problem might be temporary. T that we've successfully gotten a handle to a TCP connection!

Remember to stop the program by pressing `ctrl-c` when you're done running the code. Then restart `cargo run` after you've made each set of code chang

running the newest code.

Reading the Request

Let's implement the functionality to read the request from the browser! To start getting a connection and then taking some action with the connection, we'll write a function for processing connections. In this new `handle_connection` function, we'll read the data from the stream and print it so we can see the data being sent from the browser. Change Listing 20-2.

Filename: `src/main.rs`

```
use std::io::prelude::*;
use std::net::TcpStream;
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    println!("Request: {}", String::from_utf8_lossy(&buffer[..]));
}
```

Listing 20-2: Reading from the `TcpStream` and printing the data

We bring `std::io::prelude` into scope to get access to certain traits that let us read from the stream. In the `for` loop in the `main` function, instead of printing a message when we receive a connection, we now call the new `handle_connection` function and print the data.

In the `handle_connection` function, we've made the `stream` parameter mutable because the `TcpStream` instance keeps track of what data it returns to us internally. Instead of saving the data we ask for and save that data for the next time we ask for data, it's better to read the data as soon as we ask for it because its internal state might change; usually, we think of "reading" as not changing the state of the stream, so in this case we need the `mut` keyword.

Next, we need to actually read from the stream. We do this in two steps: first we create a buffer on the stack to hold the data that is read in. We've made the buffer 512 bytes long, which is enough to hold the data of a basic request and sufficient for our purposes if we only wanted to handle requests of an arbitrary size, buffer management would be more complicated; we'll keep it simple for now. We pass the buffer to `stream.read`, which reads from the `TcpStream` and puts them in the buffer.

Second, we convert the bytes in the buffer to a string and print that string. The `String::from_utf8_lossy` function takes a `&[u8]` and produces a `String`. The `lossy` part of the name indicates the behavior of this function when it sees an invalid UTF-8 sequence: it replaces the invalid sequence with `\u{FFFD}`, the `U+FFFD REPLACEMENT CHARACTER`. You can also use `String::from_utf8` to replace invalid characters with replacement characters for characters in the buffer that aren't filled by requests.

Let's try this code! Start the program and make a request in a web browser at `http://127.0.0.1:7878`.

get an error page in the browser, but our program's output in the terminal will look like this:

```
$ cargo run
   Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42 secs
      Running `target/debug/hello`
Request: GET / HTTP/1.1
Host: 127.0.0.1:7878
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
*****
```

Depending on your browser, you might get slightly different output. Now that we have the request data, we can see why we get multiple connections from one browser. The browser is requesting the path after `Request: GET`. If the repeated connections are all requesting the same URL, it is trying to fetch it repeatedly because it's not getting a response from our program.

Let's break down this request data to understand what the browser is asking for.

A Closer Look at an HTTP Request

HTTP is a text-based protocol, and a request takes this format:

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

The first line is the *request line* that holds information about what the client is asking for. The first part of the request line indicates the *method* being used, such as `GET` or `POST`, which tells the server what the client is making this request. Our client used a `GET` request.

The next part of the request line is `/`, which indicates the *Uniform Resource Identifier* (URI) being requested: a URI is almost, but not quite, the same as a *Uniform Resource Locator* (URL). The difference between URIs and URLs isn't important for our purposes in this chapter, so we can just mentally substitute URL for URI here.

The last part is the HTTP version the client uses, and then the request line ends with a carriage return and line feed (CRLF). CRLF stands for *carriage return* and *line feed*, which are terms from the type of sequence that separates the request line from the rest of the request data. Note that when printed, we see a new line start rather than `\r\n`.

Looking at the request line data we received from running our program so far, we can see that the method, `/` is the request URI, and `HTTP/1.1` is the version.

After the request line, the remaining lines starting from `Host:` onward are headers. The headers in this request don't have any body.

Try making a request from a different browser or asking for a different address, such as `127.0.0.1:7878/test`, to see how the request data changes.

Now that we know what the browser is asking for, let's send back some data.

Writing a Response

Now we'll implement sending data in response to a client request. Response format:

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

The first line is a *status line* that contains the HTTP version used in the response, a status code that summarizes the result of the request, and a reason phrase that provides more detail about the status code. After the CRLF sequence are any headers, another CRLF, and then the message body.

Here is an example response that uses HTTP version 1.1, has a status code of 200, no headers, and no body:

```
HTTP/1.1 200 OK\r\n\r\n
```

The status code 200 is the standard success response. The text is a tiny success message. Let's write this to the stream as our response to a successful request! From the previous function, remove the `println!` that was printing the request data and replace it with the code in Listing 20-3.

Filename: src/main.rs

```
fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];

    stream.read(&mut buffer).unwrap();

    let response = "HTTP/1.1 200 OK\r\n\r\n";
    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Listing 20-3: Writing a tiny successful HTTP response to the stream

The first new line defines the `response` variable that holds the success message. We use `as_bytes` on our `response` to convert the string data to bytes. The `write` function takes a `&[u8]` and sends those bytes directly down the connection.

Because the `write` operation could fail, we use `unwrap` on any error result. In a real application you would add error handling here. Finally, `flush` will wait for the connection to finish continuing until all the bytes are written to the connection; `TcpStream` uses a buffer to minimize calls to the underlying operating system.

With these changes, let's run our code and make a request. We're no longer running in the terminal, so we won't see any output other than the output from Cargo. When you visit `http://127.0.0.1:7878` in a web browser, you should get a blank page instead of an error message. We've coded an HTTP request and response!

Returning Real HTML

Let's implement the functionality for returning more than a blank page. Create a new file in the root of your project directory, not in the `src` directory. You can input any file name you like. Listing 20-4 shows one possibility.

Filename: hello.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

Listing 20-4: A sample HTML file to return in a response

This is a minimal HTML5 document with a heading and some text. To return when a request is received, we'll modify `handle_connection` as shown in Listing 20-5. In the HTML file, add it to the response as a body, and send it.

Filename: src/main.rs

```
use std::fs;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let contents = fs::read_to_string("hello.html").unwrap();

    let response = format!("HTTP/1.1 200 OK\r\n\r\n{}", contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Listing 20-5: Sending the contents of `hello.html` as the body of the response

We've added a line at the top to bring the standard library's `File` into scope. When reading the contents should look familiar; we used it in Chapter 12 when reading the contents of a file for our I/O project in Listing 12-4.

Next, we use `format!` to add the file's contents as the body of the success response.

Run this code with `cargo run` and load `127.0.0.1:7878` in your browser; you should see the rendered HTML!

Currently, we're ignoring the request data in `buffer` and just sending back the same HTML file unconditionally. That means if you try requesting `127.0.0.1:7878/some-page`, you'll still get back this same HTML response. Our server is very limited compared to most web servers do. We want to customize our responses depending on the URL. Instead of always sending back the HTML file for a well-formed request to `/`.

Validating the Request and Selectively Responding

Right now, our web server will return the HTML in the file no matter what the browser asks for. We can add functionality to check that the browser is requesting `/` before returning the response. If the browser requests anything else, we can return an error. For this we need to modify `handle_connection`, as shown in Listing 20-6. This new code checks the content of the request received from the browser.

know a request for / looks like and adds if and else blocks to treat requests differently.

Filename: src/main.rs

```
// --snip--

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    if buffer.starts_with(get) {
        let contents = fs::read_to_string("hello.html").unwrap();

        let response = format!("HTTP/1.1 200 OK\r\n{}\r\n", contents);
        stream.write(response.as_bytes()).unwrap();
        stream.flush().unwrap();
    } else {
        // some other request
    }
}
```

Listing 20-6: Matching the request and handling requests to / differently than other paths

First, we hardcode the data corresponding to the / request into the get variable. After reading raw bytes into the buffer, we transform get into a byte string by adding syntax at the start of the content data. Then we check whether buffer starts with it. If it does, it means we've received a well-formed request to /, which is the subject of the if block that returns the contents of our HTML file.

If buffer does not start with the bytes in get, it means we've received something else. We can add code to the else block in a moment to respond to all other requests.

Run this code now and request 127.0.0.1:7878; you should get the HTML in hello.html. For any other request, such as 127.0.0.1:7878/something-else, you'll get a connect refused error message. You saw this when running the code in Listing 20-1 and Listing 20-2.

Now let's add the code in Listing 20-7 to the else block to return a response with status code 404, which signals that the content for the request was not found. We'll also return an error page to render in the browser indicating the response to the end user.

Filename: src/main.rs

```
// --snip--

} else {
    let status_line = "HTTP/1.1 404 NOT FOUND\r\n{}\r\n";
    let contents = fs::read_to_string("404.html").unwrap();

    let response = format!("{}\r\n", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Listing 20-7: Responding with status code 404 and an error page if anything other than / is requested

Here, our response has a status line with status code 404 and the reason phrase NOT FOUND.

still not returning headers, and the body of the response will be the HTML in need to create a `404.html` file next to `hello.html` for the error page; again feel you want or use the example HTML in Listing 20-8.

Filename: `404.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

Listing 20-8: Sample content for the page to send back with any 404 responses

With these changes, run your server again. Requesting `127.0.0.1:7878` should `hello.html`, and any other request, like `127.0.0.1:7878/foo`, should return the `404.html`.

A Touch of Refactoring

At the moment the `if` and `else` blocks have a lot of repetition: they're both writing the contents of the files to the stream. The only differences are the specific filename. Let's make the code more concise by pulling out those differences in the `else` lines that will assign the values of the status line and the filename to variables that we can then use those variables unconditionally in the code to read the file and write the file. Listing 20-9 shows the resulting code after replacing the large `if` and `else` blocks.

Filename: `src/main.rs`

```
// --snip--
fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Listing 20-9: Refactoring the `if` and `else` blocks to contain only the code that handles the two cases

Now the `if` and `else` blocks only return the appropriate values for the status line and the filename as a tuple; we then use destructuring to assign these two values to `status_line` and `filename` in the `let` statement, as discussed in Chapter 18.

The previously duplicated code is now outside the `if` and `else` blocks and and `filename` variables. This makes it easier to see the difference between means we have only one place to update the code if we want to change how response writing work. The behavior of the code in Listing 20-9 will be the same as 20-8.

Awesome! We now have a simple web server in approximately 40 lines of Rust that handles one request with a page of content and responds to all other requests with a 404 error message.

Currently, our server runs in a single thread, meaning it can only serve one request at a time. In this section, we'll examine how that can be a problem by simulating some slow requests. Then we'll learn how to modify the server so it can handle multiple requests at once.

Turning Our Single-Threaded Server into a Multithreaded Server

Right now, the server will process each request in turn, meaning it won't process any other requests while it's waiting for a response. This is fine for a connection until the first is finished processing. If the server received more requests, however, serial execution would be less and less optimal. If the server receives a request that takes a long time to process, subsequent requests will have to wait until the long request has completed before they can be processed quickly. We'll need to fix this, but first, we'll look at how to simulate a slow request.

Simulating a Slow Request in the Current Server Implementation

We'll look at how a slow-processing request can affect other requests made to the server. Listing 20-10 implements handling a request to `/sleep` with a slow response that will cause the server to sleep for 5 seconds before responding.

Filename: `src/main.rs`

```
use std::thread;
use std::time::Duration;
// --snip--

fn handle_connection(mut stream: TcpStream) {
    // --snip--

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };
    // --snip--
}
```

Listing 20-10: Simulating a slow request by recognizing `/sleep` and sleeping for 5 seconds

This code is a bit messy, but it's good enough for simulation purposes. We could make it cleaner by using a `match` expression instead of `if` and `else if`.

`sleep`, whose data our server recognizes. We added an `else if` after the request to `/sleep`. When that request is received, the server will sleep for 5 seconds before sending the successful HTML page.

You can see how primitive our server is: real libraries would handle the requests in a much less verbose way!

Start the server using `cargo run`. Then open two browser windows: one for `/` and the other for `http://127.0.0.1:7878/sleep`. If you enter the `/` URI a few times, the second window will respond quickly. But if you enter `/sleep` and then load `/`, you'll see that `/` waits for its full 5 seconds before loading.

There are multiple ways we could change how our web server works to avoid getting stuck back up behind a slow request; the one we'll implement is a thread pool.

Improving Throughput with a Thread Pool

A *thread pool* is a group of spawned threads that are waiting and ready to handle tasks. When a program receives a new task, it assigns one of the threads in the pool to the task. The remaining threads in the pool are available to handle other tasks that come in while the first thread is processing. When the first thread is done processing a task, it returns to the pool of idle threads, ready to handle a new task. A thread pool allows us to handle many connections concurrently, increasing the throughput of your server.

We'll limit the number of threads in the pool to a small number to protect us from Denial of Service (DoS) attacks; if we had our program create a new thread for each request a single thread could make 10 million requests to our server could create havoc by using up all of the CPU's resources and grinding the processing of requests to a halt.

Rather than spawning unlimited threads, we'll have a fixed number of threads. When requests come in, they'll be sent to the pool for processing. The pool will manage the incoming requests. Each of the threads in the pool will pop off a request from the queue, process it, and then ask the queue for another request. With this design, we can handle many requests concurrently, where N is the number of threads. If each thread is responding to a request, subsequent requests can still back up in the queue, but we've increased the maximum number of requests we can handle before reaching that point.

This technique is just one of many ways to improve the throughput of a web server. Other techniques you might explore are the fork/join model and the single-threaded async I/O model. If you're interested in this topic, you can read more about other solutions and try to implement them in a low-level language like Rust, all of these options are possible.

Before we begin implementing a thread pool, let's talk about what using the `join` function is good for. When you're trying to design code, writing the client interface first can help you understand the API of the code so it's structured in the way you want to call it; then implement the functionality within that structure rather than implementing the functionality and then defining the interface around it.

Similar to how we used test-driven development in the project in Chapter 12, we'll use a variation of test-driven development here. We'll write the code that calls the functions we want to use, and then look at the errors from the compiler to determine what we should change next to get the code to compile.

Code Structure If We Could Spawn a Thread for Each Request

First, let's explore how our code might look if it did create a new thread for each request. As mentioned earlier, this isn't our final plan due to the problems with potentially spawning an unlimited number of threads, but it is a starting point. Listing 20-11 shows the code that would main to spawn a new thread to handle each stream within the `for` loop.

Filename: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        thread::spawn(|| {
            handle_connection(stream);
        });
    }
}
```

Listing 20-11: Spawning a new thread for each stream

As you learned in Chapter 16, `thread::spawn` will create a new thread and a closure in the new thread. If you run this code and load `/sleep` in your browser tabs, you'll indeed see that the requests to `/` don't have to wait for `/sleep`. As mentioned, this will eventually overwhelm the system because you'd be making any limit.

Creating a Similar Interface for a Finite Number of Threads

We want our thread pool to work in a similar, familiar way so switching from a thread pool doesn't require large changes to the code that uses our API. Listing 20-12 shows a hypothetical interface for a `ThreadPool` struct we want to use instead of the `thread::spawn` closure.

Filename: src/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }
}
```

Listing 20-12: Our ideal `ThreadPool` interface

We use `ThreadPool::new` to create a new thread pool with a configurable number of threads in case four. Then, in the `for` loop, `pool.execute` has a similar interface as `thread::spawn`. It takes a closure and tells the pool to run it for each stream. We need to implement `execute` to take the closure and give it to a thread in the pool to run. This code won't compile yet, so the compiler can guide us in how to fix it.

Building the `ThreadPool` Struct Using Compiler Driven Development

Make the changes in Listing 20-12 to `src/main.rs`, and then let's use the compiler to help us. Run `cargo check` to drive our development. Here is the first error we get:

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve. Use of undeclared type or module
--> src\main.rs:10:16
  |
10 |     let pool = ThreadPool::new(4);
  |     ^^^^^^^^^^^^^^^^^ Use of undeclared type or module
  |     `ThreadPool`


error: aborting due to previous error
```

Great! This error tells us we need a `ThreadPool` type or module, so we'll build one. Our `ThreadPool` implementation will be independent of the kind of work our workers do, so let's switch the `hello` crate from a binary crate to a library crate to hold our implementation. After we change to a library crate, we could also use the `seahorse` library for any work we want to do using a thread pool, not just for serving web requests.

Create a `src/lib.rs` that contains the following, which is the simplest definition of a thread pool that we can have for now:

Filename: `src/lib.rs`

```
pub struct ThreadPool;
```

Then create a new directory, `src/bin`, and move the binary crate rooted in `src/main.rs` to it. Doing so will make the library crate the primary crate in the `hello` directory. We can then move the code from `src/main.rs` to `src/bin/main.rs` using `cargo run`. After moving the `main.rs` file, we'll need to update the `Cargo.toml` file to make the library crate the primary crate in and bring `ThreadPool` into scope by adding the following code to the `[lib]` section:

Filename: `src/bin/main.rs`

```
extern crate hello;
use hello::ThreadPool;
```

This code still won't work, but let's check it again to get the next error that will tell us what we need to do.

```
$ cargo check
   Compiling hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for
`hello::ThreadPool` in the current scope
--> src/bin/main.rs:13:16
  |
13 |     let pool = ThreadPool::new(4);
  |     ^^^^^^^^^^^^^^^^^ function or associated item not
  |     found in `hello::ThreadPool`
```

This error indicates that next we need to create an associated function named `new` for the `ThreadPool` type. We also know that `new` needs to have one parameter that can accept 4 as a size and return a `ThreadPool` instance. Let's implement the simplest `new` function that matches these characteristics:

Filename: `src/lib.rs`

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

We chose `usize` as the type of the `size` parameter, because we know that threads doesn't make any sense. We also know we'll use this 4 as the number of collection of threads, which is what the `usize` type is for, as discussed in the end of Chapter 3.

Let's check the code again:

```
$ cargo check
    Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size`
--> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |           ^^^^
  |
  | = note: #[warn(unused_variables)] on by default
  | = note: to avoid this warning, consider using `_size` instead

error[E0599]: no method named `execute` found for type `hello::ThreadPool` in the current scope
--> src/bin/main.rs:18:14
  |
18 |         pool.execute(|| {
  |             ^^^^^^
```

Now we get a warning and an error. Ignoring the warning for a moment, the error tells us that we don't have an `execute` method on `ThreadPool`. Recall from the "Creating a Finite Number of Threads" section that we decided our thread pool should be similar to `thread::spawn`. In addition, we'll implement the `execute` function, which it's given and gives it to an idle thread in the pool to run.

We'll define the `execute` method on `ThreadPool` to take a closure as a parameter. This follows the "Storing Closures Using Generic Parameters and the Fn Traits" section in Chapter 10, which shows how to take closures as parameters with three different traits: `Fn`, `FnMut`, and `FnOnce`. We'll decide which kind of closure to use here. We know we'll end up doing something similar to the `thread::spawn` implementation, so we can look at what bounds the trait `FnOnce` has on its parameter. The documentation shows us the following:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static
```

The `F` type parameter is the one we're concerned with here; the `T` type parameter is the return value, and we're not concerned with that. We can see that `spawn` uses the `FnOnce` trait bound on `F`. This is probably what we want as well, because we'll eventually want to get in `execute` to `spawn`. We can be further confident that `FnOnce` is the trait we want because the thread for running a request will only execute that request's closure, and that closure matches the `Once` in `FnOnce`.

The `F` type parameter also has the trait bound `Send` and the lifetime bound `'static`. These are useful in our situation: we need `Send` to transfer the closure from one thread to another, and `'static` because we don't know how long the thread will take to execute. Let's implement the `execute` method on `ThreadPool` that will take a generic parameter of type `F` with the `FnOnce` trait bound:

Filename: `src/lib.rs`

```
impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        }
}
```

We still use the `()` after `FnOnce` because this `FnOnce` represents a closure parameters and doesn't return a value. Just like function definitions, the `return` from the signature, but even if we have no parameters, we still need the `return`:

```
$ cargo check
    Compiling hello v0.1.0 (file:///projects/hello)
warning: unused variable: `size'
--> src/lib.rs:4:16
  |
4 |     pub fn new(size: usize) -> ThreadPool {
  |           ^
  |
  = note: #[warn(unused_variables)] on by default
  = note: to avoid this warning, consider using `_size` instead

warning: unused variable: `f`
--> src/lib.rs:8:30
  |
8 |     pub fn execute<F>(&self, f: F)
  |           ^
  |
  = note: to avoid this warning, consider using `_f` instead
```

We're receiving only warnings now, which means it compiles! But note that if you make a request in the browser, you'll see the errors in the browser that we saw in the chapter. Our library isn't actually calling the closure passed to `execute` yet.

Note: A saying you might hear about languages with strict compilers, such as “if the code compiles, it works.” But this saying is not universally true. One reason is that it does absolutely nothing! If we were building a real, complete project, it would be a good time to start writing unit tests to check that the code compiles and does what we want.

Validating the Number of Threads in `new`

We'll continue to get warnings because we aren't doing anything with the parameter `execute`. Let's implement the bodies of these functions with the behavior we think about `new`. Earlier we chose an unsigned type for the `size` parameter because a negative number of threads makes no sense. However, a pool with zero threads also makes no sense, yet zero is a perfectly valid `usize`. We'll add code to check that `size` is at least one before we return a `ThreadPool` instance and have the program panic if it reaches the `assert!` macro, as shown in Listing 20-13.

Filename: `src/lib.rs`

```

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        ThreadPool
    }

    // --snip--
}

```

Listing 20-13: Implementing `ThreadPool::new` to panic if `size` is zero

We've added some documentation for our `ThreadPool` with doc comments. good documentation practices by adding a section that calls out the situation can panic, as discussed in Chapter 14. Try running `cargo doc --open` and click struct to see what the generated docs for `new` look like!

Instead of adding the `assert!` macro as we've done here, we could make `new` we did with `Config::new` in the I/O project in Listing 12-9. But we've decided to create a thread pool without any threads should be an unrecoverable error. If you're ambitious, try to write a version of `new` with the following signature to compare:

```
pub fn new(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

Creating Space to Store the Threads

Now that we have a way to know we have a valid number of threads to store, let's create those threads and store them in the `ThreadPool` struct before returning it. How do we "store" a thread? Let's take another look at the `thread::spawn` signature:

```

pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static

```

The `spawn` function returns a `JoinHandle<T>`, where `T` is the type that the closure returns. Let's use `JoinHandle` too and see what happens. In our case, the closures we're spawning will handle the connection and not return anything, so `T` will be the unit type, `()`.

The code in Listing 20-14 will compile but doesn't create any threads yet. We'll need to change the definition of `ThreadPool` to hold a vector of `thread::JoinHandle<()>` instead of `FnOnce`. We'll start by creating a vector with a capacity of `size`, set up a `for` loop that will run some code to spawn threads, and finally return a `ThreadPool` instance containing them.

Filename: `src/lib.rs`

```

use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // create some threads and store them in the vector
        }

        ThreadPool {
            threads
        }
    }
    // --snip--
}

```

Listing 20-14: Creating a vector for `ThreadPool` to hold the threads

We've brought `std::thread` into scope in the library crate, because we're using `thread::JoinHandle` as the type of the items in the vector in `ThreadPool`.

Once a valid size is received, our `ThreadPool` creates a new vector that can hold that many threads. We haven't used the `with_capacity` function in this book yet, which performs better than `Vec::new` but with an important difference: it preallocates space in the vector instead of growing it as elements are inserted. This is more efficient than using `Vec::new`, which resizes itself as elements are inserted.

When you run `cargo check` again, you'll get a few more warnings, but it should still pass.

A Worker Struct Responsible for Sending Code from the `ThreadPool` to the Thread

We left a comment in the `for` loop in Listing 20-14 regarding the creation of threads. Let's look at how we actually create threads. The standard library provides `thread::Builder::spawn` to create threads, and `thread::spawn` expects to get some code to run when the thread is created. However, in our case, we want to create the threads and have them run code that we'll send later. The standard library's implementation of threads doesn't provide a way to do this; we have to implement it manually.

We'll implement this behavior by introducing a new data structure between the `Worker` and the `ThreadPool`: a `Worker` struct that will manage the threads. We'll call this data structure `Worker`. It's a common term in pooling implementations. Think of people working in the kitchen at a restaurant; they wait until orders come in from customers, and then they're responsible for filling them.

Instead of storing a vector of `JoinHandle<()>` instances in the thread pool, each `Worker` will store a single `JoinHandle<()>` instance. We'll add a method on `Worker` that will take a closure of code to run and send it to the thread for execution. We'll also give each worker an `id` so we can distinguish between workers in the pool when logging or debugging.

Let's make the following changes to what happens when we create a `ThreadPool`. Instead of creating the code that sends the closure to the thread after we have `Worker` set up in the `new` method, we'll move that logic into the `Worker` struct.

1. Define a `Worker` struct that holds an `id` and a `JoinHandle<()>`.
2. Change `ThreadPool` to hold a vector of `Worker` instances.
3. Define a `Worker::new` function that takes an `id` number and returns a `JoinHandle<()>` that holds the `id` and a thread spawned with an empty closure.
4. In `ThreadPool::new`, use the `for` loop counter to generate an `id`, create a `Worker` that `id`, and store the worker in the vector.

If you're up for a challenge, try implementing these changes on your own before moving on to Listing 20-15.

Ready? Here is Listing 20-15 with one way to make the preceding modifications.

Filename: `src/lib.rs`

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers
        }
        // --snip--
    }
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});
        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-15: Modifying `ThreadPool` to hold `Worker` instances instead of handles

We've changed the name of the field on `ThreadPool` from `threads` to `workers`, holding `Worker` instances instead of `JoinHandle<()>` instances. We use the loop as an argument to `Worker::new`, and we store each new `Worker` in the vector.

External code (like our server in `src/bin/main.rs`) doesn't need to know the implementation details of `Worker`. It just needs to know how to regard using a `Worker` struct within `ThreadPool`, so we make the `Worker` type public.

function private. The `Worker::new` function uses the `id` we give it and store instance that is created by spawning a new thread using an empty closure.

This code will compile and will store the number of `Worker` instances we specify to `ThreadPool::new`. But we're *still* not processing the closure that we get in how to do that next.

Sending Requests to Threads via Channels

Now we'll tackle the problem that the closures given to `thread::spawn` do a Currently, we get the closure we want to execute in the `execute` method. But `thread::spawn` a closure to run when we create each `Worker` during the creation of the `ThreadPool`.

We want the `Worker` structs that we just created to fetch code to run from a `ThreadPool` and send that code to its thread to run.

In Chapter 16, you learned about *channels*—a simple way to communicate between threads. That would be perfect for this use case. We'll use a channel to function as the bridge between the `ThreadPool` and the `Worker` instances, which live in their own thread. Here is the plan:

1. The `ThreadPool` will create a channel and hold on to the sending side.
2. Each `Worker` will hold on to the receiving side of the channel.
3. We'll create a new `Job` struct that will hold the closures we want to send.
4. The `execute` method will send the job it wants to execute down the sending side of the channel.
5. In its thread, the `Worker` will loop over its receiving side of the channel and process the closures of any jobs it receives.

Let's start by creating a channel in `ThreadPool::new` and holding the sending side of the channel. The `Job` struct will hold the closures we want to send. `Job` will be the type of item we're sending down the channel.

Filename: `src/lib.rs`

```
// --snip--
use std::sync::mpsc;

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}
```

Listing 20-16: Modifying `ThreadPool` to store the sending end of a channel to

In `ThreadPool::new`, we create our new channel and have the pool hold the successfully compile, still with warnings.

Let's try passing a receiving end of the channel into each worker as the thread channel. We know we want to use the receiving end in the thread that the workers reference the `receiver` parameter in the closure. The code in Listing 20-17

Filename: src/lib.rs

```

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, receiver));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {
        let thread = thread::spawn(|| {
            receiver;
        });

        Worker {
            id,
            thread,
        }
    }
}

```

Listing 20-17: Passing the receiving end of the channel to the workers

We've made some small and straightforward changes: we pass the receiving `Worker::new`, and then we use it inside the closure.

When we try to check this code, we get this error:

```

$ cargo check
Compiling hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
--> src/lib.rs:27:42
 |
27 |         workers.push(Worker::new(id, receiver));
|                                ^^^^^^^^^^ value moved
|                                previous iteration of loop
|
= note: move occurs because `receiver` has type
`std::sync::mpsc::Receiver<Job>`, which does not implement the

```

The code is trying to pass `receiver` to multiple `Worker` instances. This won't work from Chapter 16: the channel implementation that Rust provides is multiple *consumers*. This means we can't just clone the consuming end of the channel, as we could, that is not the technique we would want to use; instead, we want to share it across threads by sharing the single `receiver` among all the workers.

Additionally, taking a job off the channel queue involves mutating the `receiver`, so we need a safe way to share and modify `receiver`; otherwise, we might get race conditions (see Chapter 16).

Recall the thread-safe smart pointers discussed in Chapter 16: to share ownership of a value between threads and allow the threads to mutate the value, we need to use `Arc<Mutex<...>>`. `Arc` will let multiple workers own the receiver, and `Mutex` will ensure that only one thread can access the receiver at a time. Listing 20-18 shows the changes we need to make.

Filename: `src/lib.rs`

```
use std::sync::Arc;
use std::sync::Mutex;
// --snip--

impl ThreadPool {
    // --snip--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }
    // --snip--
}

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --snip--
    }
}
```

Listing 20-18: Sharing the receiving end of the channel among the workers using `Arc`.

In `ThreadPool::new`, we put the receiving end of the channel in an `Arc` and give each worker a copy. In the `Worker` struct, we clone the `Arc` to bump the reference count so the workers can each have their own receiving end.

With these changes, the code compiles! We're getting there!

Implementing the `execute` Method

Let's finally implement the `execute` method on `ThreadPool`. We'll also change the type signature of `execute` to use a type alias for a trait object that holds the type of closure that `execute` receives. This is similar to the "Creating Type Synonyms with Type Aliases" section of Chapter 19, where we used type aliases to make long types shorter. Look at Listing 20-19.

Filename: `src/lib.rs`

```
// --snip--

type Job = Box<dyn FnOnce() + Send + 'static>;

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(job).unwrap();
    }
}

// --snip--
```

Listing 20-19: Creating a `Job` type alias for a `Box` that holds each closure and sends it down the channel

After creating a new `Job` instance using the closure we get in `execute`, we’re sending end of the channel. We’re calling `unwrap` on `send` for the case that happen if, for example, we stop all our threads from executing, meaning the stopped receiving new messages. At the moment, we can’t stop our threads threads continue executing as long as the pool exists. The reason we use `unwrap` in the failure case won’t happen, but the compiler doesn’t know that.

But we’re not quite done yet! In the worker, our closure being passed to `thr references` the receiving end of the channel. Instead, we need the closure to I receiving end of the channel for a job and running the job when it gets one. shown in Listing 20-20 to `Worker::new`.

Filename: src/lib.rs

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                (*job)();
            }
        });
}

Worker {
    id,
    thread,
}
```

Listing 20-20: Receiving and executing the jobs in the worker’s thread

Here, we first call `lock` on the `receiver` to acquire the mutex, and then we any errors. Acquiring a lock might fail if the mutex is in a *poisoned* state, which other thread panicked while holding the lock rather than releasing the lock. `unwrap` to have this thread panic is the correct action to take. Feel free to ch

expect with an error message that is meaningful to you.

If we get the lock on the mutex, we call `recv` to receive a `Job` from the channel. This moves past any errors here as well, which might occur if the thread holding the channel has shut down, similar to how the `send` method returns `Err` if the channel is closed.

The call to `recv` blocks, so if there is no job yet, the current thread will wait until one becomes available. The `Mutex<T>` ensures that only one `Worker` thread at a time is trying to receive a job.

Theoretically, this code should compile. Unfortunately, the Rust compiler isn't able to figure out what's going on here:

```
error[E0161]: cannot move a value of type std::ops::FnOnce() +  
std::marker::Send: the size of std::ops::FnOnce() + std::marker::Send  
is not known at this point in time  
  statically determined  
  --> src/lib.rs:63:17  
  |  
63 |         (*job)();  
  |         ^^^^^^
```

This error is fairly cryptic because the problem is fairly cryptic. To call a `FnOnce` closure in a `Box<T>` (which is what our `Job` type alias is), the closure needs to move its `self` parameter because the closure takes ownership of `self` when we call it. In general, Rust doesn't allow us to move a value out of a `Box<T>` because Rust doesn't know how big the value is. Recall in Chapter 15 that we used `Box<T>` precisely because we had some control over the size of the value that we wanted to store in a `Box<T>` to get a value of a known size.

As you saw in Listing 17-15, we can write methods that use the syntax `self: Box<T>`. This allows the method to take ownership of a `self` value stored in a `Box<T>`. This is exactly what we want to do here, but unfortunately Rust won't let us: the part of Rust that implements `FnOnce` for closures when a closure is called isn't implemented using `self: Box<dyn Self>`. So I understand that it could use `self: Box<dyn Self>` in this situation to take ownership of the closure and move the closure out of the `Box<T>`.

Rust is still a work in progress with places where the compiler could be improved. If you run the code in Listing 20-20, it should work just fine. People just like you are working on these issues! After you've finished this book, we would love for you to join in.

But for now, let's work around this problem using a handy trick. We can tell the compiler that we can take ownership of the value inside the `Box<T>` using `self: Box<T>`. Since we have ownership of the closure, we can call it. This involves defining a new method `call_box` that will use `self: Box<dyn Self>` in its signature, defining a trait `Call` that implements `FnOnce()`, changing our type alias to use the new trait, and then implementing `Call` for the closure type. These changes are shown in Listing 20-21.

Filename: `src/lib.rs`

```

trait FnBox {
    fn call_box(self: Box<dyn Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<dyn F>) {
        (*self)()
    }
}

type Job = Box<dyn FnBox + Send + 'static>;

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
        let thread = thread::spawn(move || {
            loop {
                let job = receiver.lock().unwrap().recv().unwrap();

                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });
    Worker {
        id,
        thread,
    }
}

```

Listing 20-21: Adding a new trait `FnBox` to work around the current limitation of `FnOnce`

First, we create a new trait named `FnBox`. This trait has the one method `call_box` that delegates to the `call` methods on the other `Fn*` traits except that it takes `self: Box` instead of `self` and move the value out of the `Box<T>`.

Next, we implement the `FnBox` trait for any type `F` that implements the `FnOnce` trait. This means that any `FnOnce()` closures can use our `call_box` method. The `call_box` uses `(*self)()` to move the closure out of the `Box<T>` and call it.

We now need our `Job` type alias to be a `Box` of anything that implements `FnBox`. This will allow us to use `call_box` in `Worker` when we get a `Job` value instead of a closure directly. Implementing the `FnBox` trait for any `FnOnce()` closure means we don't care what the actual values we're sending down the channel. Now Rust knows what we want to do is fine.

This trick is very sneaky and complicated. Don't worry if it doesn't make perfect sense right away. It will be completely unnecessary.

With the implementation of this trick, our thread pool is in a working state! Consider how we can make some requests:

```
$ cargo run
    Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never used: `workers'
--> src/lib.rs:7:5
  |
7 |     workers: Vec<Worker>,
  |     ^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `id`
--> src/lib.rs:61:5
  |
61 |     id: usize,
  |     ^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: field is never used: `thread`
--> src/lib.rs:62:5
  |
62 |     thread: thread::JoinHandle<()>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

Finished dev [unoptimized + debuginfo] target(s) in 0.99 secs
Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

Success! We now have a thread pool that executes connections asynchronously. Since we created more than four threads, so our system won't get overloaded if the server receives many requests. If we make a request to `/sleep`, the server will be able to serve other requests while another thread runs them.

After learning about the `while let` loop in Chapter 18, you might be wondering how the worker thread code as shown in Listing 20-22.

Filename: `src/lib.rs`

```
// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) ->
        let thread = thread::spawn(move || {
            while let Ok(job) = receiver.lock().unwrap().recv() {
                println!("Worker {} got a job; executing.", id);

                job.call_box();
            }
        });

        Worker {
            id,
            thread,
        }
    }
}
```

Listing 20-22: An alternative implementation of `Worker::new` using `while let`

This code compiles and runs but doesn't result in the desired threading behavior. It still cause other requests to wait to be processed. The reason is somewhat subtle: `lock` has no public `unlock` method because the ownership of the lock is based on the `MutexGuard<T>` returned from the `lock` method. At any time, the borrow checker can then enforce the rule that a resource guarded by a lock can only be accessed unless we hold the lock. But this implementation can also result in a longer lifetime than intended if we don't think carefully about the lifetime of the `MutexGuard`: the values in the `while` expression remain in scope for the duration of the block, held for the duration of the call to `job.call_box()`, meaning other workers

By using `loop` instead and acquiring the lock and a job within the block rather than `lock`, the `MutexGuard` returned from the `lock` method is dropped as soon as the `let` block ends. This ensures that the lock is held during the call to `recv`, but it is released before the call to `job.call_box()`, allowing multiple requests to be serviced concurrently.

Graceful Shutdown and Cleanup

The code in Listing 20-21 is responding to requests asynchronously through its `poll` method as we intended. We get some warnings about the `workers`, `id`, and `thread` variables being used in a direct way that reminds us we're not cleaning up anything. When we press `ctrl-c` to halt the main thread, all other threads are stopped immediately, even though they're in the middle of serving a request.

Now we'll implement the `Drop` trait to call `join` on each of the threads in the `workers` vector when the requests they're working on before closing. Then we'll implement a way for the `Threadpool` to stop accepting new requests and shut down. To see this code in action, run the `rust-threadpool` server to accept only two requests before gracefully shutting down its threads.

Implementing the `Drop` Trait on `ThreadPool`

Let's start with implementing `Drop` on our thread pool. When the pool is dropped, each of the threads in the `workers` vector should join to make sure they finish their work. Listing 20-23 shows a first attempt at this implementation; this code won't quite work yet.

Filename: `src/lib.rs`

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            worker.thread.join().unwrap();
        }
    }
}
```

Listing 20-23: Joining each thread when the thread pool goes out of scope

First, we loop through each of the thread pool `workers`. We use `&mut` for the mutable reference, and we also need to be able to mutate `worker`. For each message saying that this particular worker is shutting down, and then we call `thread`. If the call to `join` fails, we use `unwrap` to make Rust panic and go into shutdown.

Here is the error we get when we compile this code:

```
error[E0507]: cannot move out of borrowed content
--> src/lib.rs:65:13
 |
65 |         worker.thread.join().unwrap();
|         ^^^^^^ cannot move out of borrowed content
```

The error tells us we can't call `join` because we only have a mutable borrow. `join` takes ownership of its argument. To solve this issue, we need to move the `Worker` instance that owns `thread` so `join` can consume the thread. We do this by having `Worker` hold an `Option<thread::JoinHandle<()>` instead, we can call the `Option` to move the value out of the `Some` variant and leave a `None` variant. In other words, a `Worker` that is running will have a `Some` variant in `thread`, and when it's not a `Worker`, we'll replace `Some` with `None` so the `Worker` doesn't have a thread.

So we know we want to update the definition of `Worker` like this:

Filename: `src/lib.rs`

```
struct Worker {
    id: usize,
    thread: Option<thread::JoinHandle<()>>,
}
```

Now let's lean on the compiler to find the other places that need to change. We get two errors:

```

error[E0599]: no method named `join` found for type
`std::option::Option<std::thread::JoinHandle<()>` in the current scope
--> src/lib.rs:65:27
  |
65 |         worker.thread.join().unwrap();
  |         ^^^^^^

error[E0308]: mismatched types
--> src/lib.rs:89:13
  |
89 |             thread,
  |             ^^^^^^
  |
  |             expected enum `std::option::Option`, found struct
`std::thread::JoinHandle`
  |             help: try using a variant of the expected type: `Some(_)`
  |
= note: expected type `std::option::Option<std::thread::JoinHandle<()>`
        found type `std::thread::JoinHandle<_>`

```

Let's address the second error, which points to the code at the end of `Worker::new`. We need to wrap the `thread` value in `Some` when we create a new `Worker`. Make the following changes to fix this error:

Filename: `src/lib.rs`

```

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Self {
        // --snip--

        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

The first error is in our `Drop` implementation. We mentioned earlier that we need to move the `thread` value to move `thread` out of `worker`. The following changes fix this.

Filename: `src/lib.rs`

```

impl Drop for ThreadPool {
    fn drop(&mut self) {
        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}

```

As discussed in Chapter 17, the `take` method on `Option` takes the `Some` value and returns `None` in its place. We're using `if let` to destructure the `Some` and get the `thread` value. Then we call `join` on the `thread`. If a worker's `thread` is already `None`, we know that worker has already been cleaned up, so nothing happens in that case.

Signaling to the Threads to Stop Listening for Jobs

With all the changes we've made, our code compiles without any warnings. Excellent!

code doesn't function the way we want it to yet. The key is the logic in the child threads of the `Worker` instances: at the moment, we call `join`, but that won't work because they `loop` forever looking for jobs. If we try to drop our `Thread` current implementation of `drop`, the main thread will block forever waiting to finish.

To fix this problem, we'll modify the threads so they listen for either a `Job` to process or a `Terminate`. They should stop listening and exit the infinite loop. Instead of `Job` instances, we can use one of these two enum variants.

Filename: `src/lib.rs`

```
enum Message {
    NewJob(Job),
    Terminate,
}
```

This `Message` enum will either be a `NewJob` variant that holds the `Job` the thread needs to process, or a `Terminate` variant that will cause the thread to exit its loop and stop listening.

We need to adjust the channel to use values of type `Message` rather than `Job`. Listing 20-24.

Filename: `src/lib.rs`

```

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

// --snip--

impl ThreadPool {
    // --snip--

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

// --snip--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>>)
        -> Worker {
        let thread = thread::spawn(move ||{
            loop {
                let message = receiver.lock().unwrap().recv().unwrap();

                match message {
                    Message::NewJob(job) => {
                        println!("Worker {} got a job; executing.", id);
                        job.call_box();
                    },
                    Message::Terminate => {
                        println!("Worker {} was told to terminate.", id);
                        break;
                    },
                }
            }
        });
        Worker {
            id,
            thread: Some(thread),
        }
    }
}

```

Listing 20-24: Sending and receiving `Message` values and exiting the loop if a `Message::Terminate`

To incorporate the `Message` enum, we need to change `Job` to `Message` in the definition of `ThreadPool` and the signature of `Worker::new`. The `execute` method of `ThreadPool` will send jobs wrapped in the `Message::NewJob` variant. Then, in `Worker::new`, when a job is received from the channel, the job will be processed if the `NewJob` variant is received. If a `Terminate` variant is received, the thread will break out of the loop if the `Terminate` variant is received.

With these changes, the code will compile and continue to function in the same way as Listing 20-21. But we'll get a warning because we aren't creating any messages of the `Drop` variety. Let's fix this warning by changing our `Drop` implementation to look like this:

Filename: src/lib.rs

```
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

Listing 20-25: Sending `Message::Terminate` to the workers before calling `join` on each thread

We're now iterating over the workers twice: once to send one `Terminate` message to each worker and once to call `join` on each worker's thread. If we tried to send a message to a worker while it was still processing a request, we couldn't guarantee that the worker in the current iteration would receive the message from the channel.

To better understand why we need two separate loops, imagine a scenario where we used a single loop to iterate through each worker, on the first iteration a terminate message was sent down the channel and `join` was called on the first worker's thread. If the first worker was still processing a request at that moment, the second worker would pick up the message from the channel and shut down. We would be left waiting on the first worker to shutdown because the second thread picked up the terminate message. Deadlock!

To prevent this scenario, we first put all of our `Terminate` messages on the channel and then we join on all the threads in another loop. Each worker will stop receiving messages from the channel once it gets a terminate message. So, we can be sure that if we send terminate messages as there are workers, each worker will receive a terminate message before `join` is called on its thread.

To see this code in action, let's modify `main` to accept only two requests before shutting down the server, as shown in Listing 20-26.

Filename: src/bin/main.rs

```
fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}
```

Listing 20-26: Shut down the server after serving two requests by exiting the

You wouldn't want a real-world web server to shut down after serving only two requests, but this just demonstrates that the graceful shutdown and cleanup is in working order.

The `take` method is defined in the `Iterator` trait and limits the iteration to the first two requests. The `ThreadPool` will go out of scope at the end of `main`, and the `drop` implementation will run.

Start the server with `cargo run`, and make three requests. The third request will be dropped, so in your terminal you should see output similar to this:

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.0 secs
        Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 3 got a job; executing.
Shutting down.
Sending terminate message to all workers.
Shutting down all workers.
Shutting down worker 0
Worker 1 was told to terminate.
Worker 2 was told to terminate.
Worker 0 was told to terminate.
Worker 3 was told to terminate.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

You might see a different ordering of workers and messages printed. We can see what's happening by looking at the logs. Notice that the server accepted two requests from clients, and then the server stopped accepting connections. When the `ThreadPool` goes out of scope at the end of `main`, its `Drop` implementation kicks in, and the pool tells all workers to terminate. Each print a message when they see the terminate message, and then the main thread shuts down each worker thread.

Notice one interesting aspect of this particular execution: the `ThreadPool` sends the terminate message down the channel, and before any worker received the message, the main thread told worker 0 to finish. In the meantime, each of the workers received the terminate message. Once worker 0 finished, the main thread waited for the rest of the workers to finish. By the time the main thread had all received the termination message and were able to shut down.

Congrats! We've now completed our project; we have a basic web server that can respond asynchronously. We're able to perform a graceful shutdown of the server, even if it has many threads in the pool.

Here's the full code for reference:

Filename: `src/bin/main.rs`

```
extern crate hello;
use hello::ThreadPool;

use std::io::prelude::*;
use std::net::TcpListener;
use std::net::TcpStream;
use std::fs;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 512];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK\r\n\r\n", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND\r\n\r\n", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!("{}{}", status_line, contents);

    stream.write(response.as_bytes()).unwrap();
    stream.flush().unwrap();
}
```

Filename: src/lib.rs

```
use std::thread;
use std::sync::mpsc;
use std::sync::Arc;
use std::sync::Mutex;

enum Message {
    NewJob(Job),
    Terminate,
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Message>,
}

trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}

type Job = Box<dyn FnBox + Send + 'static>;

impl ThreadPool {
    /// Create a new ThreadPool.
    ///
    /// The size is the number of threads in the pool.
    ///
    /// # Panics
    ///
    /// The `new` function will panic if the size is zero.
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool {
            workers,
            sender,
        }
    }

    pub fn execute<F>(&self, f: F)
        where
            F: FnOnce() + Send + 'static
    {
        let job = Box::new(f);

        self.sender.send(Message::NewJob(job)).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!("Sending terminate message to all workers.");
    }
}
```

```

        for _ in &mut self.workers {
            self.sender.send(Message::Terminate).unwrap();
        }

        println!("Shutting down all workers.");

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }

    struct Worker {
        id: usize,
        thread: Option<thread::JoinHandle<()>>,
    }

    impl Worker {
        fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Message>>: Worker {
            let thread = thread::spawn(move ||{
                loop {
                    let message = receiver.lock().unwrap().recv().unwrap();

                    match message {
                        Message::NewJob(job) => {
                            println!("Worker {} got a job; executing.", id);
                            job.call_box();
                        },
                        Message::Terminate => {
                            println!("Worker {} was told to terminate.", id);
                            break;
                        },
                    }
                }
            });
            Worker {
                id,
                thread: Some(thread),
            }
        }
    }
}

```

We could do more here! If you want to continue enhancing this project, here

- Add more documentation to `ThreadPool` and its public methods.
- Add tests of the library's functionality.
- Change calls to `unwrap` to more robust error handling.
- Use `ThreadPool` to perform some task other than serving web requests instead. Then compare its API and robustness to the thread pool we im

Summary

Well done! You've made it to the end of the book! We want to thank you for joining Rust. You're now ready to implement your own Rust projects and help with community discussions. Keep in mind that there is a welcoming community of other Rustaceans who are here to help you with any challenges you encounter on your Rust journey.

Appendix

The following sections contain reference material you may find useful in your development.

Appendix A: Keywords

The following list contains keywords that are reserved for current or future language features. As such, they cannot be used as identifiers, such as names of functions, parameters, struct fields, modules, crates, constants, macros, static values, and lifetimes.

Keywords Currently in Use

The following keywords currently have the functionality described.

- `as` - perform primitive casting, disambiguate the specific trait containing items in use and `extern crate` statements
- `break` - exit a loop immediately
- `const` - define constant items or constant raw pointers
- `continue` - continue to the next loop iteration
- `crate` - link an external crate or a macro variable representing the crate being defined
- `else` - fallback for `if` and `if let` control flow constructs
- `enum` - define an enumeration
- `extern` - link an external crate, function, or variable
- `false` - Boolean false literal
- `fn` - define a function or the function pointer type
- `for` - loop over items from an iterator, implement a trait, or specify a trait bound
- `if` - branch based on the result of a conditional expression
- `impl` - implement inherent or trait functionality
- `in` - part of `for` loop syntax
- `let` - bind a variable
- `loop` - loop unconditionally
- `match` - match a value to patterns
- `mod` - define a module
- `move` - make a closure take ownership of all its captures
- `mut` - denote mutability in references, raw pointers, or pattern binding
- `pub` - denote public visibility in struct fields, `impl` blocks, or modules
- `ref` - bind by reference
- `return` - return from function
- `self` - a type alias for the type implementing a trait
- `self` - method subject or current module
- `static` - global variable or lifetime lasting the entire program execution
- `struct` - define a structure
- `super` - parent module of the current module
- `trait` - define a trait

- `true` - Boolean true literal
- `type` - define a type alias or associated type
- `unsafe` - denote unsafe code, functions, traits, or implementations
- `use` - import symbols into scope
- `where` - denote clauses that constrain a type
- `while` - loop conditionally based on the result of an expression

Keywords Reserved for Future Use

The following keywords do not have any functionality but are reserved by Rust for future use.

- `abstract`
- `alignof`
- `become`
- `box`
- `do`
- `final`
- `macro`
- `offsetof`
- `override`
- `priv`
- `proc`
- `pure`
- `sizeof`
- `typeof`
- `unsized`
- `virtual`
- `yield`

Appendix B: Operators and Symbols

This appendix contains a glossary of Rust's syntax, including operators and characters that appear by themselves or in the context of paths, generics, trait bounds, macro definitions, comments, tuples, and brackets.

Operators

Table B-1 contains the operators in Rust, an example of how the operator would be used, a short explanation, and whether that operator is overloadable. If an operator is overloadable, the relevant trait to use to overload that operator is listed.

Table B-1: Operators

Operator	Example	Explanation
!	<code>ident!(...),</code> <code>ident!{...},</code> <code>ident![...]</code>	Macro expansion
!	<code>!expr</code>	Bitwise or logical complement
<code>!=</code>	<code>var != expr</code>	Nonequality comparison

Operator	Example	Explanation
%	expr % expr	Arithmetic remainder
%=	var %= expr	Arithmetic remainder and assignment
&	&expr, &mut expr	Borrow
&	&type, &mut type, &'a type, &'a mut type	Borrowed pointer type
&	expr & expr	Bitwise AND
&=	var &= expr	Bitwise AND and assignment
&&	expr && expr	Logical AND
*	expr * expr	Arithmetic multiplication
*=	var *= expr	Arithmetic multiplication and assignment
*	*expr	Dereference
*	*const type, *mut type	Raw pointer
+	trait + trait, 'a + trait	Compound type constraint
+	expr + expr	Arithmetic addition
+=	var += expr	Arithmetic addition and assignment
,	expr, expr	Argument and element separator
-	- expr	Arithmetic negation
-	expr - expr	Arithmetic subtraction
-=	var -= expr	Arithmetic subtraction and assignment
->	fn(...) -> type, ... -> type	Function and closure return type
.	expr.ident	Member access
..	..., expr..., ..expr, expr..expr	Right-exclusive range literal
..	..expr	Struct literal update syntax
..	variant(x, ...), struct_type { x, .. }	"And the rest" pattern binding
...	expr...expr	In a pattern: inclusive range pattern
/	expr / expr	Arithmetic division
/=	var /= expr	Arithmetic division and assignment
:	pat: type, ident: type	Constraints
:	ident: expr	Struct field initializer
:	'a: loop {...}	Loop label
;	expr;	Statement and item terminator
;	[...; len]	Part of fixed-size array syntax

Operator	Example	Explanation
<code><<</code>	<code>expr << expr</code>	Left-shift
<code><=</code>	<code>var <= expr</code>	Left-shift and assignment
<code><</code>	<code>expr < expr</code>	Less than comparison
<code><=</code>	<code>expr <= expr</code>	Less than or equal to comparison
<code>=</code>	<code>var = expr, ident = type</code>	Assignment/equivalence
<code>==</code>	<code>expr == expr</code>	Equality comparison
<code>=></code>	<code>pat => expr</code>	Part of match arm syntax
<code>></code>	<code>expr > expr</code>	Greater than comparison
<code>>=</code>	<code>expr >= expr</code>	Greater than or equal to comparison
<code>>></code>	<code>expr >> expr</code>	Right-shift
<code>>>=</code>	<code>var >>= expr</code>	Right-shift and assignment
<code>@</code>	<code>ident @ pat</code>	Pattern binding
<code>^</code>	<code>expr ^ expr</code>	Bitwise exclusive OR
<code>^=</code>	<code>var ^= expr</code>	Bitwise exclusive OR and assignment
<code> </code>	<code>pat pat</code>	Pattern alternatives
<code> </code>	<code>expr expr</code>	Bitwise OR
<code> =</code>	<code>var = expr</code>	Bitwise OR and assignment
<code> </code>	<code>expr expr</code>	Logical OR
<code>?</code>	<code>expr?</code>	Error propagation

Non-operator Symbols

The following list contains all non-letters that don't function as operators; they like a function or method call.

Table B-2 shows symbols that appear on their own and are valid in a variety

Table B-2: Stand-Alone Syntax

Symbol	Explanation
<code>'ident</code>	Named lifetime or loop label
<code>...u8, ...i32, ...f64, ...usize, etc.</code>	Numeric literal of specific type
<code>"..."</code>	String literal
<code>r"...", r#"..."#, r##"..."##, etc.</code>	Raw string literal, escape characters
<code>b"..."</code>	Byte string literal; constructs a <code>[u8]</code> vector
<code>br"...", br#"..."#, br##"..."##, etc.</code>	Raw byte string literal, combinative string literal
<code>'...</code>	Character literal
<code>b'...'</code>	ASCII byte literal

Symbol	Explanation
<code> ... expr</code>	Closure
<code>!</code>	Always empty bottom type for division
<code>-</code>	"Ignored" pattern binding; also useful for literals readable

Table B-3 shows symbols that appear in the context of a path through the module item.

Table B-3: Path-Related Syntax

Symbol	Explanation
<code>ident::ident</code>	Namespace path
<code>::path</code>	Path relative to the crate root (i.e., absolute path)
<code>self::path</code>	Path relative to the current module (relative path).
<code>super::path</code>	Path relative to the parent of the current module
<code>type::ident</code> , <code><type as trait>::ident</code>	Associated constants, functions, and methods
<code><type>::...</code>	Associated item for a type that can implement it (e.g., <code><&T>::... , <[T]>::... , etc.</code>)
<code>trait::method(...)</code>	Disambiguating a method call by name if it defines it
<code>type::method(...)</code>	Disambiguating a method call by name if it's defined in another type
<code><type as trait>::method(...)</code>	Disambiguating a method call by name if it's defined in another type

Table B-4 shows symbols that appear in the context of using generic type parameters.

Table B-4: Generics

Symbol	Explanation
<code>path<...></code>	Specifies parameters to generic type in a type alias
<code>path::<...></code> , <code>method::<...></code>	Specifies parameters to generic type, function or method expression; often referred to as turbofish (e.g., <code>"42".parse::<i32>()</code>)
<code>fn ident<...> ...</code>	Define generic function
<code>struct ident<...></code> ...	Define generic structure
<code>enum ident<...> ...</code>	Define generic enumeration
<code>impl<...> ...</code>	Define generic implementation
<code>for<...> type</code>	Higher-ranked lifetime bounds
<code>type<ident=>type</code>	A generic type where one or more associate types are associated with type parameters (e.g., <code>Iterator<Item=T></code>)

Table B-5 shows symbols that appear in the context of constraining generic type parameters via trait bounds.

Table B-5: Trait Bound Constraints

Symbol	Explanation
<code>T: U</code>	Generic parameter <code>T</code> constrained to types that implement <code>U</code>
<code>T: 'a</code>	Generic type <code>T</code> must outlive lifetime <code>'a</code> (meanir transitivity)
<code>T : 'static</code>	Generic type <code>T</code> contains no borrowed references
<code>'b: 'a</code>	Generic lifetime <code>'b</code> must outlive lifetime <code>'a</code>
<code>T: ?Sized</code>	Allow generic type parameter to be a dynamically sized type
<code>'a + trait, trait + trait</code>	Compound type constraint

Table B-6 shows symbols that appear in the context of calling or defining macro attributes on an item.

Table B-6: Macros and Attributes

Symbol	Explanation
<code>#[meta]</code>	Outer attribute
<code>#![meta]</code>	Inner attribute
<code>\$ident</code>	Macro substitution
<code>\$ident:kind</code>	Macro capture
<code>\$(...)...</code>	Macro repetition

Table B-7 shows symbols that create comments.

Table B-7: Comments

Symbol	Explanation
<code>//</code>	Line comment
<code>//!</code>	Inner line doc comment
<code>///</code>	Outer line doc comment
<code>/*...*/</code>	Block comment
<code>/*!...*/</code>	Inner block doc comment
<code>/**...*/</code>	Outer block doc comment

Table B-8 shows symbols that appear in the context of using tuples.

Table B-8: Tuples

Symbol	Explanation
<code>()</code>	Empty tuple (aka unit), both literal and type
<code>(expr)</code>	Parenthesized expression
<code>(expr,)</code>	Single-element tuple expression
<code>(type,)</code>	Single-element tuple type
<code>(expr, ...)</code>	Tuple expression
<code>(type, ...)</code>	Tuple type
<code>expr(expr, ...)</code>	Function call expression; also used to create <code>struct</code> s and tuple <code>enum</code> variants

Symbol	Explanation
<code>ident!(...), ident!{...}, ident![...]</code>	Macro invocation
<code>expr.0, expr.1, etc.</code>	Tuple indexing

Table B-9 shows the contexts in which curly braces are used.

Table B-9: Curly Brackets

Context	Explanation
<code>{...}</code>	Block expression
Type <code>{...}</code>	<code>struct</code> literal

Table B-10 shows the contexts in which square brackets are used.

Table B-10: Square Brackets

Context	Explanation
<code>[...]</code>	Array literal
<code>[expr; len]</code>	Array literal containing <code>len</code> copies of <code>expr</code>
<code>[type; len]</code>	Array type containing <code>len</code> instances of <code>type</code>
<code>expr[expr]</code>	Collection indexing. Overloadable (<code>Index</code> , <code>IndexMut</code> , <code>IndexSize</code>)
<code>expr[..], expr[a..], expr[..b], expr[a..b]</code>	Collection indexing pretending to be <code>Range</code> , <code>RangeFrom</code> , <code>RangeTo</code> , or <code>RangeFull</code>

Appendix C: Derivable Traits

In various places in the book, we've discussed the `derive` attribute, which you can add to a struct, enum, or trait definition. The `derive` attribute generates code that will implement one or more traits for your type, giving it default implementation on the type you've annotated with the `derive` syntax.

In this appendix, we provide a reference of all the traits in the standard library that have a `derive` attribute. Each section covers:

- What operators and methods deriving this trait will enable
- What the implementation of the trait provided by `derive` does
- What implementing the trait signifies about the type
- The conditions in which you're allowed or not allowed to implement the trait
- Examples of operations that require the trait

If you want different behavior than that provided by the `derive` attribute, consider implementing the trait manually. See the standard library documentation for each trait for details on how to manually implement it.

The rest of the traits defined in the standard library can't be implemented or derived. These traits don't have sensible default behavior, so it's up to you to implement them in a way that makes sense for what you're trying to accomplish.

An example of a trait that can't be derived is `Display`, which handles formatting. When you implement `Display` for a type, you should always consider the appropriate way to display a type to an end user. What information should an end user be allowed to see? What parts would they find relevant? What parts would be most relevant to them? The Rust compiler doesn't have this insight, so you'll need to provide appropriate default behavior for you.

The list of derivable traits provided in this appendix is not comprehensive: it lists traits that you can derive for their own traits, making the list of traits you can use derive with `derive`. Implementing `derive` involves using a procedural macro, which is covered in the [Procedural Macros](#) chapter.

Debug for Programmer Output

The `Debug` trait enables debug formatting in format strings, which you indicate with `{}` placeholders.

The `Debug` trait allows you to print instances of a type for debugging purposes. Programmers using your type can inspect an instance at a particular point in the code.

The `Debug` trait is required, for example, in use of the `assert_eq!` macro. The macro compares two values of instances given as arguments if the equality assertion fails so program knows the two instances weren't equal.

PartialEq and Eq for Equality Comparisons

The `PartialEq` trait allows you to compare instances of a type to check for equality. It provides the `==` and `!=` operators.

Deriving `PartialEq` implements the `eq` method. When `PartialEq` is derived, instances are equal only if *all* fields are equal, and the instances are not equal if any field is not equal. When derived on enums, each variant is equal to itself and not equal to other variants.

The `PartialEq` trait is required, for example, with the use of the `assert_eq!` macro to be able to compare two instances of a type for equality.

The `Eq` trait has no methods. Its purpose is to signal that for every value of the type, the value is equal to itself. The `Eq` trait can only be applied to types that also implement `PartialEq`, although not all types that implement `PartialEq` can implement `Eq`. One exception is floating point number types: the implementation of floating point numbers states that not-a-number (`Nan`) value is not equal to each other.

An example of when `Eq` is required is for keys in a `HashMap<K, V>` so the `HashMap` can determine whether two keys are the same.

PartialOrd and Ord for Ordering Comparisons

The `PartialOrd` trait allows you to compare instances of a type for sorting purposes. Implementing `PartialOrd` can be used with the `<`, `>`, `<=`, and `>=` operators. The `PartialOrd` trait is often combined with the `PartialEq` trait to types that also implement `PartialEq`.

Deriving `PartialOrd` implements the `partial_cmp` method, which returns `Ordering::Less`, `Ordering::Greater`, or `Ordering::Equal`. It will return `None` when the values given don't produce an ordering. An example of this is the `is_nan` method on floating point numbers: it doesn't produce an ordering, even though most values of that type can be compared. Calling `partial_cmp` with any floating point number and a NaN floating point value will return `None`.

When derived on structs, `PartialOrd` compares two instances by comparing the fields in the order in which the fields appear in the struct definition. When derived on enums, the variants declared earlier in the enum definition are considered less than the variants declared later.

The `PartialOrd` trait is required, for example, for the `gen_range` method from the `rand` crate.

generates a random value in the range specified by a low value and a high value.

The `Ord` trait allows you to know that for any two values of the annotated type exist. The `Ord` trait implements the `cmp` method, which returns an `Ordering` `Option<Ordering>` because a valid ordering will always be possible. You can add to types that also implement `PartialOrd` and `Eq` (and `Eq` requires `PartialOrd`). Structs and enums, `cmp` behaves the same way as the derived implementation does with `PartialOrd`.

An example of when `Ord` is required is when storing values in a `BTreeSet<T>`. It stores data based on the sort order of the values.

Clone and Copy for Duplicating Values

The `Clone` trait allows you to explicitly create a deep copy of a value, and the implementation might involve running arbitrary code and copying heap data. See the “Ways to Interact: Clone” section in Chapter 4 for more information on `Clone`.

Deriving `Clone` implements the `clone` method, which when implemented for a type `T` calls `clone` on each of the parts of the type. This means all the fields or values in the type must implement `Clone` to derive `Clone`.

An example of when `Clone` is required is when calling the `to_vec` method on a type that doesn't own the type instances it contains, but the vector returned from `to_vec` instances, so `to_vec` calls `clone` on each item. Thus, the type stored in the vector must implement `Clone`.

The `Copy` trait allows you to duplicate a value by only copying bits stored on the stack if no heap data is necessary. See the “Stack-Only Data: Copy” section in Chapter 4 for more information on `Copy`.

The `Copy` trait doesn't define any methods to prevent programmers from overriding methods and violating the assumption that no arbitrary code is being run. This means that programmers can assume that copying a value will be very fast.

You can derive `Copy` on any type whose parts all implement `Copy`. You can add the `Copy` trait to types that also implement `Clone`, because a type that implements `Clone` must implement the `Copy` trait to have the same implementation of `Clone` that performs the same task as `Copy`.

The `Copy` trait is rarely required; types that implement `Copy` have optimizations built in so you don't have to call `clone`, which makes the code more concise.

Everything possible with `Copy` you can also accomplish with `Clone`, but the `Copy` trait has some optimizations built in so you don't have to use `clone` in places.

Hash for Mapping a Value to a Value of Fixed Size

The `Hash` trait allows you to take an instance of a type of arbitrary size and map it to a value of fixed size using a hash function. Deriving `Hash` implements the `hash` method, and the implementation of the `hash` method combines the result of calling `hash` on all the fields or values in the type, meaning all fields or values must also implement `Hash` to derive `Hash`.

An example of when `Hash` is required is in storing keys in a `HashMap<K, V>`.

Default for Default Values

The `Default` trait allows you to create a default value for a type. Deriving `Default` provides a `default` function. The derived implementation of the `default` function calls `Default::default` on each part of the type, meaning all fields or values in the type must also implement `Default`.

The `Default::default` function is commonly used in combination with the `Option` type, as discussed in the “Creating Instances From Other Instances With Struct Updaters” section of Chapter 5. You can customize a few fields of a struct and then set and use a default value for the rest of the fields by using `..Default::default()`.

The `Default` trait is required when you use the method `unwrap_or_default` on `Option` instances, for example. If the `Option<T>` is `None`, the method `unwrap_or_default` returns the result of `Default::default` for the type `T` stored in the `Option<T>`.

Appendix D: Macros

We’ve used macros like `println!` throughout this book but haven’t fully explained how they work or how it works. This appendix explains macros as follows:

- What macros are and how they differ from functions
- How to define a declarative macro to do metaprogramming
- How to define a procedural macro to create custom `derive` traits

We’re covering the details of macros in an appendix because they’re still evolving and changing rapidly. The details of how macros work have changed and, in the near future, will change at a quicker rate than the rest of the standard library since Rust 1.0, so this section is more likely to become out-of-date over time. Due to Rust’s stability guarantees, the code shown here will continue to work across major versions, but there may be additional capabilities or easier ways to write macros available at the time of this publication. Bear that in mind when you try to implement your own macros in this appendix.

The Difference Between Macros and Functions

Fundamentally, macros are a way of writing code that writes other code, which is called *metaprogramming*. In Appendix C, we discussed the `derive` attribute, which generates the implementation of various traits for you. We’ve also used the `println!` and `format!` macros throughout the book. All of these macros *expand* to produce more code than you would write manually.

Metaprogramming is useful for reducing the amount of code you have to write, but it is also one of the roles of functions. However, macros have some additional benefits that functions don’t have.

A function signature must declare the number and type of parameters the function takes. On the other hand, a macro can take a variable number of parameters: we can call `print` with one argument or `println!("hello {}")`, with two arguments. Also, macros are evaluated at compile time, while functions are evaluated at runtime. The compiler interprets the meaning of the code, so a macro can, for example, generate code that depends on the type of its arguments. A function can’t, because it gets called at runtime and a trait needs to be implemented at runtime.

The downside to implementing a macro instead of a function is that macro code is more complex than function definitions because you’re writing Rust code that writes Rust code.

this indirection, macro definitions are generally more difficult to read, understand than function definitions.

Another difference between macros and functions is that macro definitions are within modules like function definitions are. To prevent unexpected name clashes with external crates, you have to explicitly bring the macros into the scope of your module as you bring the external crate into scope, using the `#[macro_use]` attribute. An example would bring all the macros defined in the `serde` crate into the scope of your module:

```
#[macro_use]
extern crate serde;
```

If `extern crate` was able to bring macros into scope by default without this attribute, it would be prevented from using two crates that happened to define macros with the same name. In practice, this conflict doesn't occur often, but the more crates you use, the more likely it is to happen.

There is one last important difference between macros and functions: you must bring macros into scope *before* you call them in a file, whereas you can define functions anywhere.

Declarative Macros with `macro_rules!` for General Metaprogramming

The most widely used form of macros in Rust are *declarative macros*. These are also referred to as *macros by example*, *macro_rules!* *macros*, or just plain *macros*. Declarative macros allow you to write something similar to a Rust `match` expression. Chapter 6, `match` expressions are control structures that take an expression and compare its value to patterns, and then run the code associated with the pattern that matches. Macros also compare a value to patterns that have code associated with the pattern. If the value is the literal Rust source code passed to the macro, the patterns are code snippets of that source code, and the code associated with each pattern is the code passed to the macro. This all happens during compilation.

To define a macro, you use the `macro_rules!` construct. Let's explore how to use this construct by looking at how the `vec!` macro is defined. Chapter 8 covered how we can use `macro_rules!` to create a new vector with particular values. For example, the following macro definition creates a vector with three integers inside:

```
let v: Vec<u32> = vec![1, 2, 3];
```

We could also use the `vec!` macro to make a vector of two integers or a vector of four integers. We wouldn't be able to use a function to do the same because we wouldn't know how many values up front.

Let's look at a slightly simplified definition of the `vec!` macro in Listing D-1.

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing D-1: A simplified version of the `vec!` macro definition

Note: The actual definition of the `vec!` macro in the standard library includes code to preallocate the correct amount of memory up front. That code is an optimization and is included here to make the example simpler.

The `#[macro_export]` annotation indicates that this macro should be made available in the crate in which we're defining the macro is imported. Without this annotation, if the macro is used in another crate, depending on this crate uses the `#[macro_use]` annotation, the macro would be visible in the current crate's scope.

We then start the macro definition with `macro_rules!` and the name of the macro, `vec!`, without the exclamation mark. The name, in this case `vec`, is followed by curly braces, which enclose the body of the macro definition.

The structure in the `vec!` body is similar to the structure of a `match` expression with a single arm, where the pattern is `($($x:expr),*)`, followed by `=>` and the block of code associated with this pattern. If the pattern matches, the associated block of code will be emitted. Because there is only one pattern in this macro, there is only one valid way to match; any other macro with more than one pattern will have more than one arm.

Valid pattern syntax in macro definitions is different than the pattern syntax in regular expressions because macro patterns are matched against Rust code structure rather than text. This section explains through what the pieces of the pattern in Listing D-1 mean; for the full macro pattern syntax, see the [macro pattern reference](#).

First, a set of parentheses encompasses the whole pattern. Next comes a comma, followed by another set of parentheses, which captures values that match the pattern within the replacement code. Within `$()` is `$x:expr`, which matches any Rust expression. The `$x` is replaced by the name `$x`.

The comma following `$()` indicates that a literal comma separator character appears after the code that matches the code captured in `$()`. The `*` following the comma indicates that the pattern matches zero or more of whatever precedes the `*`.

When we call this macro with `vec![1, 2, 3];`, the `$x` pattern matches three expressions `1`, `2`, and `3`.

Now let's look at the pattern in the body of the code associated with this arm. The code within the `$()*` part is generated for each part that matches `$()` in the pattern, times depending on how many times the pattern matches. The `$x` is replaced by the value matched. When we call this macro with `vec![1, 2, 3];`, the code generated by the macro call will be the following:

```
let mut temp_vec = Vec::new();
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
temp_vec
```

We've defined a macro that can take any number of arguments of any type and create a vector containing the specified elements.

Given that most Rust programmers will *use* macros more than *write* macros, `macro_rules!` any further. To learn more about how to write macros, consult documentation or other resources, such as ["The Little Book of Rust Macros"](#).

Procedural Macros for Custom `derive`

The second form of macros is called *procedural macros* because they're more like a function than a macro (they're a type of procedure). Procedural macros accept some Rust code as an input, and produce some Rust code as an output rather than matching against it and replacing the code with other code as declarative macros do. At the time of this writing, procedural macros are still experimental, so you'll need to define procedural macros to allow your traits to be implemented on a type by adding the trait name in a `derive` annotation.

We'll create a crate named `hello_macro` that defines a trait named `HelloMacro` and a function named `hello_macro`. Rather than making our crate users implement the trait for each of their types, we'll provide a procedural macro so users can annotate their code with `#[derive(HelloMacro)]` to get a default implementation of the `hello_macro` trait. This implementation will print `Hello, Macro! My name is TypeName!` where `TypeName` is the type on which this trait has been defined. In other words, we'll write a crate that makes it easy for a programmer to write code like Listing D-2 using our crate.

Filename: `src/main.rs`

```
extern crate hello_macro;
#[macro_use]
extern crate hello_macro_derive;

use hello_macro::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Listing D-2: The code a user of our crate will be able to write when using our procedural macro.

This code will print `Hello, Macro! My name is Pancakes!` when we're done defining the trait. Let's start by creating a new library crate, like this:

```
$ cargo new hello_macro --lib
```

Next, we'll define the `HelloMacro` trait and its associated function:

Filename: `src/lib.rs`

```
pub trait HelloMacro {
    fn hello_macro();
}
```

We have a trait and its function. At this point, our crate user could implement desired functionality, like so:

```
extern crate hello_macro;

use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

However, they would need to write the implementation block for each type that implements the `HelloMacro`; we want to spare them from having to do this work.

Additionally, we can't yet provide a default implementation for the `hello_macro` trait. If we did, it would print the name of the type the trait is implemented on: Rust doesn't have references to types at runtime, so we can't look up the type's name at runtime. We need a macro to generate code for us.

The next step is to define the procedural macro. At the time of this writing, procedural macros can only be in their own crate. Eventually, this restriction might be lifted. The current workflow for creating a procedural macro crate and macro crates is as follows: for a crate named `foo`, a custom derive macro for `bar` is called `foo_derive`. Let's start a new crate called `hello_macro_derive` to support the `hello_macro` project:

```
$ cargo new hello_macro_derive --lib
```

Our two crates are tightly related, so we create the procedural macro crate `hello_macro_derive` inside our `hello_macro` crate. If we change the trait definition in `hello_macro`, we will also change the implementation of the procedural macro in `hello_macro_derive` as well. This way, the procedural macro crate can be published separately, and programmers using these crates will need to add both of them as dependencies and bring them both into scope. We could instead have the `hello_macro` crate publish the procedural macro as a dependency, but this structured the project makes it possible for programmers to use `hello_macro` without having to add both crates as dependencies if they only want the `derive` functionality.

We need to declare the `hello_macro_derive` crate as a procedural macro crate and add the procedural macro functionality from the `syn` and `quote` crates, as you'll see in a moment, so we will add them as dependencies. Add the following to the `Cargo.toml` file for `hello_macro_derive`:

Filename: `hello_macro_derive/Cargo.toml`

```
[lib]
proc-macro = true

[dependencies]
syn = "0.11.11"
quote = "0.3.15"
```

To start defining the procedural macro, place the code in Listing D-3 into your `hello_macro_derive` crate. Note that this code won't compile until we add an `impl_hello_macro` function.

Filename: `hello_macro_derive/src/lib.rs`

```
extern crate proc_macro;
extern crate syn;
#[macro_use]
extern crate quote;

use proc_macro::TokenStream;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a string representation of the type definition
    let s = input.to_string();

    // Parse the string representation
    let ast = syn::parse_derive_input(&s).unwrap();

    // Build the impl
    let gen = impl_hello_macro(&ast);

    // Return the generated impl
    gen.parse().unwrap()
}
```

Listing D-3: Code that most procedural macro crates will need to have for procedural macros.

Notice the way we've split the functions in D-3; this will be the same for almost every procedural macro crate you see or create, because it makes writing a procedural macro easier: you choose to do in the place where the `impl_hello_macro` function is called depending on your procedural macro's purpose.

We've introduced three new crates: `proc_macro`, `syn`, and `quote`. The `proc_macro` crate is part of the standard library, so we didn't need to add that to the dependencies in `Cargo.toml`. The `syn` crate allows us to convert Rust code into a string containing that Rust code. The `quote` crate converts that string back into Rust code. These crates make it easy to parse any sort of Rust code we might want to handle: writing a full parser for a language like C is a much more difficult task.

The `hello_macro_derive` function will get called when a user of our library adds `#[derive(HelloMacro)]` on a type. The reason is that we've annotated the `hello_macro_derive` function here with `proc_macro_derive` and specified the name, `HelloMacro`, as the trait name; that's the convention most procedural macros follow.

This function first converts the `input` from a `TokenStream` to a `String` by calling `to_string`. `String` is a string representation of the Rust code for which we are deriving a trait. In the example in Listing D-2, `s` will have the `String` value `struct Pancakes;` because that was the code we added the `#[derive(HelloMacro)]` annotation to.

Note: At the time of this writing, you can only convert a `TokenStream` to a `String`. Support for other types will exist in the future.

Now we need to parse the Rust code `String` into a data structure that we can perform operations on. This is where `syn` comes into play. The `parse_derive_input` function in `syn` takes a `String` and returns a `DeriveInput` struct representing the parsed Rust code.

following code shows the relevant parts of the `DeriveInput` struct we get from `Pancakes`:

```
DeriveInput {
    // --snip--

    ident: Ident(
        "Pancakes"
    ),
    body: Struct(
        Unit
    )
}
```

The fields of this struct show that the Rust code we've parsed is a unit struct (identifier, meaning the name) of `Pancakes`. There are more fields on this struct of Rust code; check the [syn documentation for `DeriveInput`](#) for more.

At this point, we haven't defined the `impl_hello_macro` function, which is where the Rust code we want to include. But before we do, note that the last part of this function uses the `parse` function from the `quote` crate to turn the output of the function back into a `TokenStream`. The returned `TokenStream` is added to the code users write, so when they compile their crate, they'll get extra functionality there.

You might have noticed that we're calling `unwrap` to panic if the calls to the `parse` functions fail here. Panicking on errors is necessary in procedural macros; procedural macro derive functions must return `TokenStream` rather than `Result`. We've chosen to simplify this example by using `unwrap`; you should provide more specific error messages about what went wrong by `expect`.

Now that we have the code to turn the annotated Rust code from a `TokenStream` into a `DeriveInput` instance, let's generate the code that implements the `HelloMacro` annotated type:

Filename: `hello_macro_derive/src/lib.rs`

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> quote::Tokens {
    let name = &ast.ident;
    quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    }
}
```

We get an `Ident` struct instance containing the name (identifier) of the annotated type, `ast.ident`. The code in Listing D-2 specifies that the `name` will be `Ident("Pancakes")`.

The `quote!` macro lets us write the Rust code that we want to return and convert it into a `TokenStream`. This macro also provides some very cool templating mechanisms: `#name`, and `quote!` will replace it with the value in the variable named `name`. This is repetition similar to the way regular macros work. Check out the [quote crate introduction](#).

We want our procedural macro to generate an implementation of our `HelloMacro` trait for the user annotated, which we can get by using `#name`. The trait implementation for `HelloMacro`, whose body contains the functionality we want to provide: printing "Hello, Macro! My name is Pancakes".

Hello, Macro! My name is and then the name of the annotated type.

The `stringify!` macro used here is built into Rust. It takes a Rust expression at compile time turns the expression into a string literal, such as `"1 + 2"`. `T` `format!` or `println!`, which evaluate the expression and then turn the result into a string. It's also possible that the `#name` input might be an expression to print literally. Using `stringify!` also saves an allocation by converting `#name` to a string!

At this point, `cargo build` should complete successfully in both `hello_macro` and `hello_macro_derive`. Let's hook up these crates to the code in Listing D-2 to see them in action! Create a new binary project in your `projects` directory using `cargo new hello_macro_app`. We need to add `hello_macro` and `hello_macro_derive` as dependencies in `Cargo.toml`. If you're publishing your versions of `hello_macro` and `hello_macro_derive` to <https://crates.io/>, they would be regular dependencies; if not, you can specify them as follows:

```
[dependencies]
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Put the code from Listing D-2 into `src/main.rs`, and run `cargo run`: it should output `Hello, Macro! My name is Pancakes!`. The implementation of the `HelloMacro` procedural macro was included without the `pancakes` crate needing to implement `#[derive(HelloMacro)]` added the trait implementation.

The Future of Macros

In the future, Rust will expand declarative and procedural macros. Rust will have a declarative macro system with the `macro_rules!` keyword and will add more types of procedural macros that can do more powerful tasks than just `derive`. These systems are still under development and are not yet published; please consult the online Rust documentation for the latest information.

Appendix E: Translations of the Book

For resources in languages other than English. Most are still in progress; see the [Translations page](#) for help or let us know about a new translation!

- [Português \(BR\)](#)
- [Português \(PT\)](#)
- [Tiếng việt](#)
- [简体中文, alternate](#)
- [Українська](#)
- [Español](#)
- [Italiano](#)
- [Русский](#)
- [한국어](#)
- [日本語](#)
- [Français](#)
- [Polski](#)
- [עברית](#)
- [Cebuano](#)
- [Tagalog](#)

Appendix F - How Rust is Made and Rust"

This appendix is about how Rust is made and how that affects you as a Rust

Stability Without Stagnation

As a language, Rust cares a *lot* about the stability of your code. We want a foundation you can build on, and if things were constantly changing, that would be the same time, if we can't experiment with new features, we may not find out about them after their release, when we can no longer change things.

Our solution to this problem is what we call "stability without stagnation", and this: you should never have to fear upgrading to a new version of stable Rust. It should be painless, but should also bring you new features, fewer bugs, and

Choo, Choo! Release Channels and Riding the Trains

Rust development operates on a *train schedule*. That is, all development is done on a branch of the Rust repository. Releases follow a software release train mode used by Cisco IOS and other software projects. There are three *release channels* for

- Nightly
- Beta
- Stable

Most Rust developers primarily use the stable channel, but those who want to try new features may use nightly or beta.

Here's an example of how the development and release process works: let's say the team is working on the release of Rust 1.5. That release happened in December, and provide us with realistic version numbers. A new feature is added to Rust: a branch. Each night, a new nightly version of Rust is produced. Every six weeks, and these releases are created by our release infrastructure automatically. So here's what releases look like this, once a night:

```
nightly: * - - * - - *
```

Every six weeks, it's time to prepare a new release! The beta branch of the repository is branched off from the master branch used by nightly. Now, there are two releases:

```
nightly: * - - * - - *
          |
beta:      *
```

Most Rust users do not use beta releases actively, but test against beta in the meantime to help Rust discover possible regressions. In the meantime, there's still a nightly release:

```
nightly: * - - * - - * - - * - - *
          |
beta:      *
```

Let's say a regression is found. Good thing we had some time to test the beta release! The fix is applied to master, so that results in:

the fix is backported to the `beta` branch, and a new release of beta is produced:

```
nightly: * - - * - - * - - * - - * - - *
|
beta:      * - - - - - - - *
```

Six weeks after the first beta was created, it's time for a stable release! The `stable` branch is produced from the `beta` branch:

```
nightly: * - - * - - * - - * - - * - - * - - *
|
beta:      * - - - - - - - *
|
stable:    *
```

Hooray! Rust 1.5 is done! However, we've forgotten one thing: because the `stable` branch is six weeks behind the `beta` branch, we also need a new beta of the *next* version of Rust, 1.6. So after `stable` branches off of `beta`, it branches off of `nightly` again:

```
nightly: * - - * - - * - - * - - * - - * - - *
|
beta:      * - - - - - - - *
|
stable:    *
```

This is called the "train model" because every six weeks, a release "leaves the station" and takes a journey through the beta channel before it arrives as a stable release.

Rust releases every six weeks, like clockwork. If you know the date of one Rust release, you can calculate the date of the next one: it's six weeks later. A nice aspect of having releases every six weeks is that the next train is coming soon. If a feature happens to miss a particular release, there's no need to worry: another one is happening in a short time! This helps reduce the chance of shipping possibly unpolished features in close to the release deadline.

Thanks to this process, you can always check out the next build of Rust and try it out. It's easy to upgrade to: if a beta release doesn't work as expected, you can roll back to the previous one and get it fixed before the next stable release happens! Breakage in a beta release is still a piece of software, and bugs do exist.

Unstable Features

There's one more catch with this release model: unstable features. Rust uses "feature flags" to determine what features are enabled in a given release. If a feature is experimental or still under active development, it lands on `master`, and therefore, in `nightly`, but behind the scenes. If a user, wish to try out the work-in-progress feature, you can, but you must build your code with `rustc` and annotate your source code with the appropriate flag to opt in.

If you're using a beta or stable release of Rust, you can't use any feature flag that's not included in the release. This is a good thing: it allows us to get practical use with new features before we declare them stable. If you're a developer who wishes to opt into the bleeding edge, you can do so, and those who want a rock-solid experience can stick with stable and know that their code won't break. Stability without stagnation!

This book only contains information about stable features, as in-progress features are experimental and surely they'll be different between when this book was written and when it's published. You can find documentation for nightly-only features online.

Rustup and the Role of Rust Nightly

Rustup makes it easy to change between different release channels of Rust, project basis. By default, you'll have stable Rust installed. To install nightly, fc

```
$ rustup install nightly
```

You can see all of the *toolchains* (releases of Rust and associated component with `rustup` as well. Here's an example on one of your authors' Windows cc

```
> rustup toolchain list
stable-x86_64-pc-windows-msvc (default)
beta-x86_64-pc-windows-msvc
nightly-x86_64-pc-windows-msvc
```

As you can see, the stable toolchain is the default. Most Rust users use stable might want to use stable most of the time, but use nightly on a specific project about a cutting-edge feature. To do so, you can use `rustup override` in the set the nightly toolchain as the one `rustup` should use when you're in that directory.

```
$ cd ~/projects/needs-nightly
$ rustup override set nightly
```

Now, every time you call `rustc` or `cargo` inside of `~/projects/needs-nightly`, you know that you are using nightly Rust, rather than your default of stable Rust. This is especially useful if you have a lot of Rust projects!

The RFC Process and Teams

So how do you learn about these new features? Rust's development model follows the *Comments (RFC) process*. If you'd like an improvement in Rust, you can write up an RFC.

Anyone can write RFCs to improve Rust, and the proposals are reviewed and voted on by the Rust team, which is comprised of many topic subteams. There's a full list of the teams here, which includes teams for each area of the project: language design, compiler infrastructure, documentation, and more. The appropriate team reads the proposal, comments, writes some comments of their own, and eventually, there's a consensus to either accept or reject the feature.

If the feature is accepted, an issue is opened on the Rust repository, and someone starts working on it. The person who implements it very well may not be the person who proposed it! When the implementation is ready, it lands on the `master` branch below the discussion thread we discussed in the "Unstable Features" section.

After some time, once Rust developers who use nightly releases have been using the feature, team members will discuss the feature, how it's worked out on nightly, and whether it should make it into stable Rust or not. If the decision is to move forward, the feature is removed from nightly, and the feature is now considered stable! It rides the trains into a stable release of Rust.