Mastering the Rust Programming Language

This book provides a comprehensive guide to the Rust programming language, covering all essential topics from installation to advanced programming concepts. It is suitable for both beginners and experienced programmers looking to deepen their understanding of Rust, its unique features, and best practices.

Table of Contents

Foreword - The Rust Programming Language

- Auto
- Light
- Rust
- Coal
- Navy
- Ayu

The Rust Programming Language

Foreword

It wasn't always so clear, but the Rust programming language is fundamentally about *empowerment*: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program 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 is seen as arcane, accessible only to a select few who have devoted the necessary years learning to avoid its infamous pitfalls. And even those who practice it do so with caution, lest their code be open to exploits, crashes, or corruption.

Rust breaks down these barriers by eliminating the old pitfalls and providing a friendly, polished set of tools to help you along the way. Programmers who need to "dip down" into lower-level control can do so with Rust, without taking on the customary risk of crashes or security holes, and without having to learn the fine points of a fickle toolchain. Better yet, the language is designed to guide you 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 raise their ambitions. For example, introducing parallelism in Rust is a relatively low-risk operation: the compiler will catch the classical mistakes for you. And you can tackle more aggressive optimizations in your code with the confidence that you won't accidentally introduce crashes or vulnerabilities.

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

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

— Nicholas Matsakis and Aaron Turon

Not Found

Looks like you've taken a wrong turn.
Some things that might be helpful to you though:

§ Search

- From the Standard Library
- From DuckDuckGo

§ Reference

- The Rust official site
- The Rust reference

§ Docs

- The standard library

Installation - The Rust Programming Language

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Hello

Hello

Hello

Hello

Hello

Hello, Cargo! - The Rust Programming Language

Introduction to Cargo

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

The simplest Rust programs, like the one we've written so far, don't have any dependencies. If we had built the "Hello, world!" project with Cargo, it would only use the part of Cargo that handles building your code. As you write more complex Rust programs, you'll add dependencies, and if 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 that you're using Cargo too. Cargo comes installed with Rust if you used the official installers discussed in the "Installation" section. If you installed Rust through some other means, check whether Cargo is installed by entering the following in your terminal:

```
$ cargo --version
```

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

Creating a Project with Cargo

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

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

The first command creates a new directory and project called *hello_cargo*. We've named our project *hello_cargo*, and 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 generated two files and one directory for us: a `Cargo.toml` file and a `src` directory with a `main.rs` file inside.

It has also initialized a new Git repository along with a `.gitignore` file. Git files won't be generated if you run `cargo new` within an existing Git repository; you can override this behavior by using `cargo new --vcs=git`.

> **Note:** Git is a common version control system. You can change `cargo new` 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 code in Listing 1-2.

Filename: Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2024"

[dependencies]
```

This file is in the TOML (*Tom's Obvious, Minimal Language*) format, which is Cargo's configuration format.

The first line, `[package]`, is a section heading that indicates that the following statements are configuring a 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 your program: the name, the version, and the edition of Rust to use. We'll talk about the `edition` key in Appendix E.

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

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 Listing 1-1! So far, the differences between our project and the project Cargo generated are that Cargo placed the code in the `src` directory and we have a `Cargo.toml` configuration file in the top directory.

Cargo expects your source files to live inside the `src` directory. The top-level project directory is just for README files, license information, configuration files, and anything else not related to your code. Using Cargo helps you organize your projects. There's a place for everything, and everything is in its place.

If you started a project that doesn't use Cargo, as we did with the "Hello, world!" project, you can convert it to a project that does use Cargo. Move the project code into the `src` directory and create an appropriate `Cargo.toml` file. One easy way to get that `Cargo.toml` file is to run `cargo init`, which will create it for you automatically.

Building and Running a Cargo Project

Now let's look at what's different when we build and run the "Hello, world!" program with Cargo! From 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.exe` on Windows) rather than in your current directory. Because the default build is a debug build, Cargo puts the binary in a directory named `debug`. You can run the executable with this command:

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

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

We just built a project with `cargo build` and ran it with `./target/debug/hello_cargo`, but we can also use `cargo run` to compile the code and then run the resultant executable all in one command:

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

Using `cargo run` is more convenient than having to remember to run `cargo build` and then use the whole path to the binary, so most developers use `cargo run`.

Notice that this time we didn't see output indicating that Cargo was compiling `hello_cargo`. Cargo figured out that the files hadn't changed, so it didn't rebuild but just ran the binary. If you had modified your source code, Cargo would have rebuilt the project before running it, and you would have seen this 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 your code to make sure it compiles but doesn't produce an executable:

```
$ cargo check
   Checking 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 than `cargo build` because it skips the step of producing an executable. If you're continually checking your work while writing the code, using `cargo check` will speed up the process of letting you know if your project is still compiling! As such, many Rustaceans run `cargo check` periodically as they write their program to make sure it compiles. Then they run `cargo build` when they're ready to use the executable.

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

- We can create a project using `cargo new`.
- We can build a project using `cargo build`.
- We can build and run a project in one step using `cargo run`.
- We can build a project without producing a binary to check for errors using `cargo check`.
- Instead of saving the result of the build in the same directory as our code, Cargo stores it in the `target/debug` directory.

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

Building for Release

When your project is finally ready for release, you can use `cargo build --release` to compile it with optimizations. This command will create an executable in `target/release` instead of `target/debug`. 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, when you want to rebuild quickly and often, and another for building the final program you'll give to a user that won't be rebuilt repeatedly and that will run as fast as possible. If you're benchmarking your code's running time, be sure to run `cargo build --release` and benchmark with the executable in `target/release`.

Cargo as Convention

With simple projects, Cargo doesn't provide a lot of value over just using `rustc`, but it will prove its worth as your programs become more intricate. Once programs grow to multiple files or need a dependency, it's much easier to let Cargo coordinate the build.

Even though the `hello_cargo` project is simple, it now uses much of the real tooling you'll use in the rest of your Rust career. To work on any existing projects, you can use the following commands to check out the code using Git, change to that project's directory, and build:

```
$ git clone example.org/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 and writing Rust code. So, in Chapter 2, we'll build a guessing game program. If you would rather start by learning how common programming concepts work in Rust, see Chapter 3 and then return to Chapter 2.

Programming a Guessing Game - The Rust Programming Language

Introduction

Let's jump into Rust by working through a hands-on project together! This chapter introduces you to a few common Rust concepts by showing you how to use them in a real program. You'll learn about `let`, `match`, methods, associated functions, external crates, and more! In the following chapters, we'll explore these ideas in more detail. In this chapter, you'll just practice the fundamentals.

We'll implement a classic beginner programming problem: a guessing game. Here's how it works: the program will generate a random integer between 1 and 100. It will then prompt the player to enter a guess. After a guess is entered, the program will indicate whether the guess is too low or too high. If 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 make 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 the first argument. The second command changes to the new project's directory.

Look at the generated `Cargo.toml` file:

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2024"

[dependencies]
```

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

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

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

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] in 0.08s
     Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you need to rapidly iterate on a project, as we'll do in this game, 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 that input, and check that the input is in the expected form. To start, we'll allow the player to input a guess. Enter the code in Listing 2-1 into `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}");
}
```

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

```
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}");
}
```

By default, Rust has a set of items defined in the standard library that it brings into the scope of every program. This set is called the *prelude*, and you can see everything in it in the standard library documentation.

If a type you want to use 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 features, including the ability to accept user input.

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

```
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}");
}
```

The `fn` syntax declares a new function; the parentheses, `()`, indicate there are no parameters; 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 screen:

```
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}");
}
```

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

Storing Values with Variables

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

```
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}");
}
```

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

```
let apples = 5;
```

This line creates a new variable named `apples` and binds it to the value 5. In Rust, variables are immutable by default, meaning once we give the variable a value, the value won't change. We'll be discussing this concept in detail in the "Variables and Mutability" section in Chapter 3. To make a variable mutable, we add `mut` before the variable name:

```
let apples = 5; // immutable
let mut bananas = 5; // mutable
```

> Note: The `//` syntax starts a comment that continues until the end of the line. Rust ignores everything in comments. We'll discuss comments in more detail in Chapter 3.

Returning to the guessing game program, you now know that `let mut guess` will introduce a mutable variable named `guess`. The equal sign (`=`) tells Rust we want to bind something to the variable now. On the right of the equal sign is the value that `guess` is bound to, which is the result of calling `String::new`, a function that returns a new instance of a `String`. `String` is a string type provided 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 of the `String` type. An *associated function* is a function that's implemented on a type. This `new` function creates a new, empty string. You'll find a `new` function on many types because it's a common name for a function that makes a new value of some kind.

In full, the line:

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

has created a mutable variable that is currently bound to a new, empty instance of a `String`. Whew!

Receiving User Input

Recall that we included the input/output functionality from the standard library with `use std::io;` on the first line of the program. Now we'll call the `stdin` function from the `io` module, which will allow us to handle user input:

```
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}");
}
```

If we hadn't imported the `io` module with `use std::io;` at the beginning of the program, we could still use the function by writing `std::io::stdin`. The `stdin` function returns an instance of `std::io::Stdin`, which represents a handle to the standard input for your terminal.

Next, the line `.read_line(&mut guess)` calls the `read_line` method on the standard input handle to get input from the user. We're passing `&mut guess` as the argument to `read_line` to specify the string where the input should be stored. The full job of `read_line` is to take whatever the user types into standard input and append that into a string. The string argument needs to be mutable so the method can change the string's content.

The `&` indicates that this argument is a *reference*, providing a way to let multiple parts of your code access one piece of data without needing to copy it into memory multiple times. References are a complex feature, and one of Rust's major advantages is how safe and easy it is to use references. You don't need to know many details to finish this program. For now, just remember that, like variables, references are immutable by default. Hence, you need to write `&mut guess` rather than `&guess` to make it mutable. (Chapter 4 will explain references more thoroughly.)

Handling Potential Failure with `Result`

We're still working on this line of code. The next part is this method:

```
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}");
}
```

We could have written this 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. It's often wise to introduce a newline and other whitespace to help break up long lines when you call a method with the `.method_name()` syntax. Now let's discuss what this line does.

As mentioned earlier, `read_line` puts whatever the user enters into the string we pass to it, but it also returns a `Result` value. `Result` is an *enumeration* (enum), which is a type that can be in one of multiple states, called *variants*.

Chapter 6 will cover enums in more detail. The purpose of these `Result` types is to encode error-handling information.

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

Values of the `Result` type, like values of any type, have methods on them. An instance of `Result` has an `expect` method that you can call. If this `Result` instance is an `Err` value, `expect` will cause the program to crash and display the message you provided as an argument. If `read_line` returns an `Err`, it might be due to an error from the underlying operating system. If it is an `Ok`, `expect` will return the contained value (the number of bytes in the input) so you can use it.

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

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
   = note: `#[warn(unused_must_use)]` on by default
help: use `let _ = ...` to ignore the resulting value
   |
10 |     let _ = io::stdin().read_line(&mut guess);
   |     +++++++
```

Rust warns that you haven't handled the `Result` returned from `read_line`, indicating that the program doesn't handle a possible error.

The proper way to suppress this warning is to add error handling. In our case, we prefer the program to crash if an error occurs, so we use `expect`:

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

You'll learn about recovering from errors in Chapter 9.

Printing Values with `println!` Placeholders

Aside from the closing curly bracket, there's only one more line to discuss in the code so far:

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

This line prints the string that now contains the user's input. The `{}` set of curly brackets is a placeholder: think of `{}` as little crab pincers that hold a value in place. When printing the value of a variable, the variable name can go inside the curly brackets. When printing the result of evaluating an expression, place empty curly brackets in the format string, then follow it with a comma-separated list of expressions to print in each placeholder. Printing a variable and the result of an expression in one call looks like this:

```
#![allow(unused)]
fn main() {
    let x = 5;
    let y = 10;
    println!("x = {x} and y + 2 = {}", y + 2);
}
```

This would print `x = 5 and y + 2 = 12`.

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] in 6.44s
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

Now the first part is working: it gets input from the keyboard and prints it.

Generating a Secret Number

Next, we need to generate a secret number for the user to guess. It should be different every time for variability, so we'll use a random number between 1 and 100. Rust doesn't include random number functionality in its standard library, but the rand crate provides this functionality.

Using a Crate to Get More Functionality

A crate is a collection of Rust source code files. The project we've been building is a *binary crate* (executable). The `rand` crate is a *library crate* (a library). Cargo manages external crates very efficiently:

Before we write code that uses `rand`, we need to modify `Cargo.toml` to include it as a dependency. Add the following to `Cargo.toml` under `[dependencies]`:

```
[dependencies]
rand = "0.8.5"
```

Cargo will then fetch and compile this crate and its dependencies.

Now, build the project:

```
$ cargo build
```

This command will download the `rand` crate, compile it, and build your project with it. When you run `cargo build` again without changes, it will skip recompiling dependencies that haven't changed.

In `src/main.rs`, include code to generate a random number:

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

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

    let secret_number = rand::thread_rng().gen_range(1..=100);

    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}");
}
```

Run the program multiple times to see different secret numbers. Remove the `println!` line that outputs the secret number in the final version, because it ruins the game.

Comparing the Guess to the Secret Number

Now that we have user input and a random number, compare them as shown in Listing 2-4:

```
use std::cmp::Ordering;
use std::io;

use rand::Rng;

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

    let secret_number = rand::thread_rng().gen_range(1..=100);
```

```
    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");

    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!");
            break;
        }
    }
}
```

This code compares the user's guess with the secret number using `cmp`, which returns an `Ordering`. The `match` expression handles the three cases: less, greater, and equal. When the guess matches the secret number, it declares victory and exits the loop.

Allowing Multiple Guesses with Looping

Add a `loop` to permit multiple guesses:

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

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

    let secret_number = rand::thread_rng().gen_range(1..=100);

    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 = 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!");
                break;
            }
        }
    }
}
```

Within the loop, the game repeatedly prompts for guesses. To handle non-number inputs gracefully, modify the parsing to ignore errors:

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

This way, if parsing fails, the program simply prompts again without crashing.

Quitting After a Correct Guess

Use `break` after a correct guess to exit the loop:

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

Handling Invalid Input

Replace the `expect` call with a `match` to ignore invalid inputs:

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

Now, the game ignores non-number inputs and continues prompting.

Completing the Game

Remove the secret number output for a fair game. The final code:

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

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

    let secret_number = rand::thread_rng().gen_range(1..=100);
```

```
    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;
            }
        }
    }
}
```

Congratulations! You've successfully built the guessing game.

Summary

This project introduced many Rust concepts: variables, `match`, functions, use of external crates, and more. In upcoming chapters, these concepts will be explored in depth, including ownership, structs, enums, and more.

Variables and Mutability - The Rust Programming Language

Introduction

As mentioned in the "Storing Values with Variables" section, by default, variables are immutable. This is one of many nudges Rust gives you to write your code in a way that takes advantage of the safety and easy concurrency that Rust offers. However, you still have the option to make your variables mutable. Let's explore how and why Rust encourages you to favor immutability and why sometimes you might want to opt out.

Immutable Variables

When a variable is immutable, once a value is bound to a name, you can't change that value. To illustrate this, generate a new project called *variables* in your *projects* directory by using `cargo new variables`.

Then, in your new *variables* directory, open *src/main.rs* and replace its code with the following, which 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 regarding an immutability error, as shown:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --&gt; src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 5;
  |         +++
```

This example shows how the compiler helps you find errors in your programs. Compiler errors can be frustrating, but they only mean your program isn't safely doing what you want it to do yet; they do *not* mean that you're not a good programmer! Experienced Rustaceans still get compiler errors.

You received the error message `cannot assign twice to immutable variable` x`` because you tried to assign a second value to the immutable `x` variable.

Importance of Compile-Time Errors for Immutability

It's important to get compile-time errors when trying to change a value designated as immutable because this can lead to bugs. If one part of our code operates on the assumption that a value will never change and another part changes that value, it many not do what was designed. The Rust compiler guarantees that when you state that a value won't change, it really won't, making code easier to reason about.

Mutability

Mutability can be very useful and make code more convenient to write. Although variables are immutable by default, you can make them mutable by adding `mut` in front of the variable name as you did in Chapter 2. Adding `mut` also signals to future readers that the variable's value will change.

For example, 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 you run the program now, you get:

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

We're allowed to change the value bound to `x` from `5` to `6` because `mut` is used. Deciding whether to use mutability depends on what is clearest in a particular context.

Constants

Like immutable variables, *constants* are values bound to a name and are not allowed to change, but there are differences:

- You cannot use `mut` with constants—they're always immutable.
- Declared with the `const` keyword and require explicit type annotations.
- Can be declared in any scope, including global scope.
- Must be set to a constant expression, not a value computed at runtime.

Example:

```
#![allow(unused)]
fn main() {
    const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
}
```

Constants are useful for values needed throughout the program, such as maximum scores or physical constants.

Shadowing

You can declare a new variable with the same name as a previous variable, which "shadows" the first. Rust treats the new variable as the one in scope, effectively replacing the previous.

Example:

Filename: src/main.rs

```
fn main() {
    let x = 5;
    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

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

Outputs:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

Shadowing differs from `mut` because the latter allows changing the same variable without creating a new one, but shadowing can also change the variable's type.

For example:

```
fn main() {
    let spaces = "   ";
    let spaces = spaces.len();
}
```

Here, `spaces` first holds a string, then re-binds as its length (a number). Using `mut` to change types is not allowed:

```
fn main() {
    let mut spaces = "   ";
    spaces = spaces.len(); // Error: mismatched types
}
```

This enforces type safety and clarity.

Conclusion

Variables in Rust can be immutable, mutable, or shadowed, each with specific use cases to maintain safety and clarity. Constants serve for values that truly never change, across the entire program duration. Understanding these distinctions helps write safer, more predictable Rust code.

Data Types - The Rust Programming Language

Introduction

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

Keep in mind that Rust is a *statically typed* language, which means that it must know the types of all variables at compile time. The compiler can usually infer what type we want to use based on the value and how we use it. In cases when many types are possible, such as when we converted a `String` to a numeric type using `parse` in the "Comparing the Guess to the Secret Number" section in Chapter 2, we must add a type annotation, like this:

```
#![allow(unused)]
fn main() {
    let guess: u32 = "42".parse().expect("Not a number!");
}
```

If we don't add the `: u32` type annotation shown in the preceding code, Rust will display the following error, which means the compiler needs more information from us to know which type we want to use:

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |
  = note: cannot satisfy `<_ as FromStr>::Err == _`
help: consider giving `guess` an explicit type
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due to 1 pr
```

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, floating-point numbers, 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 in Chapter 2, the `u32` type. This type declaration indicates that the value it's associated with should be an unsigned integer (signed integer types start with `i` instead of `u`) that takes up 32 bits of space. Table 3-1 shows the built-in integer types in Rust. We can use any of these variants to declare the type of an integer value.

Table 3-1: Integer Types in Rust

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

Each variant can be either signed or unsigned and has an explicit size. *Signed* and *unsigned* refer to whether it's possible for the number to be negative—in other words, whether the number needs to have a sign with it (signed) or whether the number will only ever be positive and can therefore be represented without a sign (unsigned). It's like writing numbers on paper: when the sign matters, a number is shown with a plus sign or a minus sign; however, when it's safe to assume the number is positive, it's shown with no sign. Signed numbers are stored using two's complement representation.

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 −128 to 127. Unsigned variants can store numbers from 0 to `2^n - 1`, so a `u8` can store numbers from 0 to 255.

Additionally, the `isize` and `usize` types depend on the architecture of the computer your program is running on, which is denoted in the table as "arch": 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 number literals that can be multiple numeric types allow a type suffix, such as `57u8`, to designate the type. Number literals can also use `_` as a visual separator to make the number easier to read, such as `1_000`, which will have the same value as if you had specified `1000`.

Table 3-2: Integer Literals in Rust

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

So how do you know which type of integer to use? If you're unsure, Rust's defaults are generally good places to start: integer types default to `i32`. The primary situation in which you'd use `isize` or `usize` is when indexing some sort of collection.

Integer Overflow

Let's say you have a variable of type `u8` that can hold values between 0 and 255. If you try to change the variable to a value outside that range, such as 256, *integer overflow* will occur, which can result in one of two behaviors. When you're compiling in debug mode, Rust includes checks for integer overflow that cause your program to *panic* at runtime if this behavior occurs. Rust uses the term *panicking* when a program exits with

an error; we'll discuss panics in more depth in the "Unrecoverable Errors with `panic!`" section in Chapter 9.

When you're compiling in release mode with the `--release` flag, Rust does *not* include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs *two's complement wrapping*. In short, values greater than the maximum value the type can hold "wrap around" to the minimum of the values the type can hold. In the case of a `u8`, the value 256 becomes 0, the value 257 becomes 1, and so on. The program won't panic, but the variable will have a value that probably isn't what you were expecting it to have. Relying on integer overflow's wrapping behavior is considered an error.

To explicitly handle the possibility of overflow, you can use these families of methods provided by the standard library for primitive numeric types:

- Wrap in all modes with the `wrapping_*` methods, such as `wrapping_add`.
- Return the `None` value if there is overflow with the `checked_*` methods.
- Return the value and a Boolean indicating whether there was overflow with the `overflowing_*` methods.
- Saturate at the value's minimum or maximum values with the `saturating_*` methods.

Floating-Point Types

Rust also has two primitive types for *floating-point numbers*, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively. The default type is `f64` because on modern CPUs, it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

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.

Numeric Operations

Rust supports the basic mathematical operations you'd expect for all the number types: addition, subtraction, multiplication, division, and remainder. Integer division truncates toward zero to the nearest integer. The following code shows how you'd use each numeric operation 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;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

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

The Boolean Type

As in most other programming languages, a Boolean type in Rust has two possible values: `true` and `false`. Booleans are one byte in size. 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 use Boolean values is through conditionals, such as an `if` expression. We'll cover how `if` expressions work in Rust in the "Control Flow" section.

The Character Type

Rust's `char` type is the language's most primitive alphabetic type. Here are some examples of declaring `char` values:

```
// Filename: src/main.rs
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Note that we specify `char` literals with single quotes, as opposed to string literals, which use double quotes. Rust's `char` type is four bytes in size and represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII. Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces are all valid `char` values in Rust. Unicode Scalar Values range from `U+0000` to `U+D7FF` and `U+E000` to `U+10FFFF` inclusive. However, a "character" isn't really a concept in Unicode, so your human intuition for what a "character" is may not match up with what a `char` is in Rust. We'll discuss this topic in detail in "Storing UTF-8 Encoded Text with Strings" in Chapter 8.

Compound Types

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

The Tuple Type

A *tuple* is a general way of grouping together a number of values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.

We create a tuple by writing a comma-separated list of values inside parentheses. Each position in 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 compound element. To get the individual values out of a tuple, we can use pattern matching to destructure a 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 a pattern with `let` 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`, which is `6.4`.

We can also access a tuple element directly by 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 the tuple `x` and then accesses each element of the tuple using their respective indices. As with most programming languages, the first index in a tuple is 0.

The tuple without any values has a special name, *unit*. This value and its corresponding type are both written `()`, and represent an empty value or an empty return type. Expressions implicitly return the unit value if they don't return any other value.

The Array Type

Another way to have a collection of multiple values is with an *array*. Unlike a tuple, every element of an array must have the same type. Unlike arrays in some other languages, arrays in Rust have a fixed length.

We write the values in an array as a comma-separated list inside square brackets:

```
// 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, the same as the other types we have seen so far, rather than the heap (see Chapter 4) or when you want to ensure you always have a fixed number of elements. An array isn't as flexible as the vector type, though. A *vector* is a similar collection type provided by the standard library that *is* allowed to grow or shrink in size. If you're unsure whether to use an array or a vector, chances are you should use a vector. Chapter 8 discusses vectors in more detail.

However, arrays are more useful when you know the number of elements will not need to change. For example, if you were using the names of the months in a program, you would probably use an array rather than a vector because you know it will always contain 12 elements:

```
#![allow(unused)]
fn main() {
    let months = ["January", "February", "March", "April", "May", "June", "July",
                  "August", "September", "October", "November", "December"];
}
```

You write an array's type using square brackets with the type of each element, a semicolon, and then the number of elements in the array, like so:

```
#![allow(unused)]
fn main() {
    let a: [i32; 5] = [1, 2, 3, 4, 5];
}
```

Here, `i32` is the type of each element. After the semicolon, the number `5` indicates the array contains five elements.

You can also initialize an array to contain the same value for each element by specifying the initial value, followed by a semicolon, and then the length of the array in square brackets, as shown here:

```
#![allow(unused)]
fn main() {
    let a = [3; 5];
}
```

The array named `a` will contain 5 elements that will all be set to the value `3` initially. This is the same as writing `let a = [3, 3, 3, 3, 3];` but in a more concise way.

Accessing Array Elements

An array is a single chunk of memory of a known, fixed size that can be allocated on the stack. You can access elements of an array 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 `first` will get the value `1` because that is the value at index `[0]` in the array. The variable `second` will get the value `2` from index `[1]` in the array.

Invalid Array Element Access

Let's see what happens if you try to access an element of an array that is past the end of the array. Say you run this code, similar to the guessing game in Chapter 2, to get an array index from the user:

```
use std::io;

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

    println!("Please enter an array index.");

    let mut index = String::new();

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

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

This code compiles successfully. If you run this code using `cargo run` and enter `0`, `1`, `2`, `3`, or `4`, the program will print out the corresponding value at that index in the array. If you instead enter a number past the end of the array, such as `10`, you'll see output like this:

```
thread 'main' panicked at src/main.rs:19:19:
index out of bounds: the len is 5 but the index is 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

The program resulted in a *runtime* error at the point of using an invalid value in the indexing operation. The program exited with an error message and didn't execute the final `println!` statement. When you attempt to access an element using indexing, Rust will check that the index you've specified is less than the array length. If the index is greater than or equal to the length, Rust will panic. This check has to happen at runtime, especially in this case, because the compiler can't possibly know what value a user will enter when they run the code later.

This is an example of Rust's memory safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing. Chapter 9 discusses more of Rust's error handling and how you can write readable, safe code that neither panics nor allows invalid memory access.

Functions - The Rust Programming Language

Introduction

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

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

Filename: src/main.rs

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

    another_function();
}

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

We define a function in Rust by entering `fn` followed by a function name and a set of parentheses. 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 inside the `main` function. Note that we defined `another_function` *after* the `main` function in the source code; we could have defined it before as well. Rust doesn't care where you define your functions, only that they're defined somewhere in a scope that can be seen by the caller.

Let's start a new binary project named *functions* to explore functions further. Place the `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] in 0.28s
     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.

Parameters

We can define functions to have *parameters*, which are special variables that are part of a function's signature. When a function has parameters, you can provide it with concrete values for those parameters. Technically, the concrete values are called *arguments*, but in casual conversation, people tend to use the words *parameter*

and *argument* interchangeably for either the variables in a function's definition or the concrete values passed in when you call a function.

In this version of `another_function`, we add a parameter:

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] in 1.21s
     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 specified as `i32`. When we pass `5` into `another_function`, the `println!` macro places `5` where the pair of curly brackets containing `x` was in the format string.

In function signatures, you *must* declare the type of each parameter. This is a deliberate decision in Rust's design: requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what type you mean. The compiler is also able to give more helpful error messages if it knows what types the function expects.

When defining multiple parameters, separate the parameter declarations with commas, like this:

Filename: src/main.rs

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

This example creates a function named `print_labeled_measurement` with two parameters. The first parameter is named `value` and is an `i32`. The second is named `unit_label` and is of type `char`. The function then prints text containing both the `value` and the `unit_label`.

Let's try running this code. Replace the program currently in your *functions* project's `src/main.rs` file with the preceding example and run it using `cargo run`. The output will be:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] in 0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

Because we called the function with `5` as the value for `value` and `'h'` as the value for `unit_label`, the program output contains those values.

Statements and Expressions

Function bodies are made up of a series of statements optionally ending in an expression. So far, the functions we've covered haven't included an ending expression, but you have seen an expression as part of a statement. 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 what statements and expressions are and how their differences affect the bodies of functions.

- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value. Let's look at some examples.

We've actually already used statements and expressions. Creating a variable and assigning a value to it with the `let` keyword is a statement. In Listing 3-1, `let y = 6;` is a statement.

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

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

Function definitions are also statements; the entire preceding example is a statement in itself. (As we will see below, *calling* a function is not a statement.)

Statements do not return values. Therefore, you can't assign a `let` statement to another variable, 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 see is:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |
```

```
warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` due to 1 previous error; 1 warning emitted
```

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

Expressions evaluate to a value and make up most of the rest of the code that you'll write in Rust. Consider a math operation, such as `5 + 6`, which is an expression that evaluates to the value `11`. Expressions can be part of statements: in Listing 3-1, the `6` in the statement `let y = 6;` is an expression that evaluates to the value `6`. Calling a function is an expression. Calling a macro is an expression. A new scope block created with curly brackets is an expression, for example:

Filename: src/main.rs

```
fn main() {
    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 `let` statement. Note that the `x + 1` line doesn't have a semicolon at the end, which is unlike most of the lines you've seen so far. Expressions do not include ending semicolons. If you add a semicolon to the end of an expression, you turn it into a statement, and it will then not return a value. Keep this in mind 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 must declare their type after an arrow (`->`). In Rust, the return value of the function is synonymous with the value of the final expression in the body of the function. You can return early from a function by using the `return` keyword and specifying a value, but most functions return the last expression implicitly. Here's an example:

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—just the number `5` by itself. That's a perfectly valid function in Rust. Note that the function's return type is specified as `-> i32`. Try running this code; the output should be:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] in 0.30s
     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`. Let's examine this further. There are two important bits:

First, the line `let x = five();` shows that we're using the return value of a function to initialize a variable. Because `five` returns `5`, the line is equivalent to:

```
#![allow(unused)]
fn main() {
    let x = 5;
}
```

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

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 at the end of 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 the following error:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
```

```
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value
```

The core issue is that the function `plus_one` is expected to return an `i32`, but because the last line ends with a semicolon, it returns `()`, the unit type, instead. Removing the semicolon fixes this.

This completes the overview of functions in Rust, including parameters, statements, expressions, and return values.

Comments - The Rust Programming Language

Comments

All programmers strive to make their code easy to understand, but sometimes extra explanation is warranted. In these cases, programmers leave *comments* in their source code that the compiler will ignore but people reading the source code may find useful.

Here's a simple comment:

```
#![allow(unused)]
fn main() {
    // hello, world
}
```

In Rust, the idiomatic comment style starts a comment with two slashes, and the comment continues until the end of the line. For comments that extend beyond a single line, you'll need to include `//` on each line, like this:

```
#![allow(unused)]
fn main() {
    // So we're doing something complicated here, long enough that we need
    // multiple lines of comments to do it! Whew! Hopefully, this comment will
    // explain 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 above the 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 discuss in the "Publishing a Crate to Crates.io" section of Chapter 14.

Control Flow - The Rust Programming Language

Introduction

The ability to run some code depending on whether a condition is *true* and to run 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 of Rust code are `if` expressions and loops.

`if` Expressions

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

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

```
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`, followed by a condition. In this case, the condition checks whether or not the variable `number` has a value less than 5. We place the block of code to execute if the condition is `true` immediately after the condition inside curly brackets. Blocks of code associated with the conditions in `if` expressions are sometimes called *arms*, just like the arms in `match` expressions discussed in the "Comparing the Guess to the Secret Number" section of Chapter 2.

Optionally, we can include an `else` expression, which we chose here, to give the program an alternative block of code to execute should the condition evaluate to `false`. If you don't provide an `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` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was true
```

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

```
fn main() {
    let number = 7;

    if number < 5 {
```

```
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

Run the program again, and observe:

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

It's important to note that the condition in this code **must** be a `bool`. Rust does not automatically convert non-Boolean types to a Boolean, unlike languages such as Ruby and JavaScript. For example, the following code causes an error:

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

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

The error indicates that `number` evaluates to an integer, not a `bool`.

To check if a number is not equal to `0` and run code only then, you can write:

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

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

Handling Multiple Conditions with `else if`

You can combine multiple conditions using `else if` expressions:

```
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");
    }
}
```

The program checks each condition in turn and executes the first one that evaluates to `true`. For number `6`, the output will be:

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

Once a condition is found `true`, the remaining conditions are not checked.

Using `if` in a `let` Statement

Because `if` is an expression, you can assign its result to a variable:

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

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

This will output:

```
The value of number is: 5
```

Note that blocks evaluate to the last expression in the block. All `if` arms must return the same type; mismatched types cause errors:

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

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

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

This code results in a compile-time error because the arms return incompatible types (`i32` vs `&str`).

Repetition with Loops

Loops are used to execute code repeatedly.

Repeating code with `loop`

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

This program runs indefinitely until manually interrupted (e.g., by pressing `ctrl` - `c`). You can use `break` to exit a loop explicitly. You can also use `continue` to skip to the next iteration.

Returning values from loops

A loop can return a value via `break`. For example:

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

    let result = loop {
        counter += 1;

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

    println!("The result is {result}");
}
```

This will output:

```
The result is 20
```

Loop labels

When you have nested loops, `break` and `continue` apply to the innermost loop. Loop labels allow specifying the target loop:

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

Output:

```
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
```

```
remaining = 10
End count = 2
```

Conditional loops with `while`

`while` loops evaluate a condition each time, executing the loop body while the condition is `true`:

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

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

        number -= 1;
    }

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

Looping Through Collections

Using a `while` loop is error-prone for collections, since it requires manual index management. Instead, Rust's `for` loop is safe and concise:

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

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

This will print all elements safely and efficiently.

Counting down with `for` and `rev()`

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

Summary

You have learned about variables, data types, functions, comments, `if` expressions, and loops. To practice, consider building programs to:

- Convert temperatures between Fahrenheit and Celsius
- Generate the *n*th Fibonacci number
- Print the lyrics of "The Twelve Days of Christmas"

Next, we will discuss a Rust concept that is unique compared to many other languages: **ownership**.

What Is Ownership? - The Rust Programming Language

The Stack and the Heap

Many programming languages don't require you to think about the stack and the heap very often. But in a systems programming language like Rust, whether a value is on the stack or the heap affects how the language behaves and why you have to make certain decisions. Parts of ownership will be described in relation to the stack and the heap later in this chapter, so here is a brief explanation in preparation.

Both the stack and the heap are parts of memory available to your code to use at runtime, but they are structured in different ways. The stack stores values in the order it gets them and removes the values in the opposite order. This is referred to as *last in, first out*. Think of a stack of plates: when you add more plates, you put them on top of the pile, and when you need a plate, you take one off the top. Adding or removing plates from the middle or bottom wouldn't work as well! Adding data is called *pushing onto the stack*, and removing data is called *popping off the stack*. All data stored on the stack must have a known, fixed size. Data with an unknown size at compile time or a size that might change must be stored on the heap instead.

The heap is less organized: when you put data on the heap, you request a certain amount of space. The memory allocator finds an empty spot in the heap that is big enough, marks it as being in use, and returns a *pointer*, which is the address of that location. This process is called *allocating on the heap* and is sometimes abbreviated as just *allocating* (pushing values onto the stack is not considered allocating). Because the pointer to the heap is a known, fixed size, you can store the pointer on the stack, but when you want the actual data, you must follow the pointer. Think of being seated at a restaurant. When you enter, you state the number of people in your group, and the host finds an empty table that fits everyone and leads you there. If someone in your group comes late, they can ask where you've been seated to find you.

Pushing to the stack is faster than allocating on the heap because the allocator never has to search for a place to store new data; that location is always at the top of the stack. Comparatively, allocating space on the heap requires more work because the allocator must first find a big enough space to hold the data and then perform bookkeeping to prepare for the next allocation.

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 jump around less in memory. Continuing the analogy, consider a server at a restaurant taking orders from many tables. 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 B, then A again, and then B again would be much slower. By the same token, a processor can do its job better if it works on data that's close to other data (on the stack) rather than farther away (on the heap).

When your code calls a function, the values passed into the function (including, potentially, 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 so you don't run out of space are all problems that ownership addresses. Once you understand ownership, you won't need to think about the stack and the heap very often, but knowing that the main purpose of ownership is to manage heap data 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 work through the examples that illustrate them:

- Each value in Rust has an *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Variable Scope

Now that we're past basic Rust syntax, we won't include all the `fn main() {` code in examples, so if you're following along, make sure to put the following examples inside a `main` function manually. As a result, our 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 scope is the range within a program for which an item is valid. Take the following variable:

```
#![allow(unused)]
fn main() {
    let s = "hello";
}
```

The variable `s` refers to a string literal, where the value of the string is hardcoded into the text of our program. The variable is valid from the point at which it's declared until the end of the current *scope*. Listing 4-1 shows a program with comments annotating where the variable `s` would be valid.

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

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

In other words, there are two important points:

- 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 similar to that in other programming languages. Now we'll build on 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 those we covered in the "Data Types" section of Chapter 3. The types covered previously are of a known size, can be stored on the stack and popped off the stack when their scope is over, and can be quickly and trivially copied to make a new, independent instance if another part of code needs to use the same value in a different scope. But we want to look at data that is stored on the heap and explore how Rust knows when to clean up that data, and the `String` type is a great example.

We'll concentrate on the parts of `String` that relate to ownership. These aspects also apply to other complex data types, whether they are provided by the standard library or created by you. We'll discuss `String` in more depth in Chapter 8.

We've already seen string literals, where a string value is hardcoded into our program. String literals are convenient, but they aren't suitable for every situation where we may want to use text. One reason is that they're immutable. Another is that not every string value can be known when we write our code: for example, what if we want to take user input and store it? For these situations, Rust has a second string type, `String`. This type manages data allocated on the heap and, as such, is able to store an amount of text that is unknown at compile time. You can create a `String` from a string literal using the `from` function, like so:

```
#![allow(unused)]
fn main() {
    let s = String::from("hello");
}
```

The double colon `::` operator allows us to namespace this particular `from` function under the `String` type rather than using some sort of name like `string_from`. We'll discuss this syntax more in the Chapter 5 section "Method Syntax," and when we talk about namespacing with modules in Chapter 7.

This kind of string *can* be mutated:

```
fn main() {
    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, why can `String` be mutated but literals cannot? The difference is in 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 hardcoded directly into the final executable. This is why string literals are fast and efficient. But these properties only come from the string literal's immutability. Unfortunately, we can't put a blob of memory into the binary for each piece of text whose size is unknown at compile time and whose size might change while running the program.

With the `String` type, to support a mutable, growable piece of text, we need to allocate an amount of memory on the heap, unknown at compile time, to hold the contents. This means:

- The memory must be requested from the memory allocator at runtime.
- We need a way of returning this memory to the allocator when we're done with our `String`.

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

However, the second part is different. In languages with a *garbage collector (GC)*, the GC keeps track of and cleans up memory that isn't being used anymore, and we don't need to think about it. In most languages without a GC, it's our responsibility to identify when memory is no longer being used and to call code to explicitly free 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 have an invalid variable. If we do it twice, that's a bug too. We need to pair exactly one *allocate* with exactly one *free*.

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

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

```
        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
}
```

There is a natural point at which we can return the memory our `String` needs to the allocator: when `s` goes out of scope. When a variable goes out of scope, Rust calls a special function for us. This function is called drop, and it's where the author of `String` can put the code to return the memory. Rust calls `drop` automatically at the closing curly bracket.

Note:

In C++, this pattern of deallocating resources at the end of an item's lifetime is sometimes called *Resource Acquisition Is Initialization (RAII)*. The `drop` function in Rust will be familiar to you if you've used RAII patterns.

This pattern has a profound impact on the way Rust code is written. It may seem simple right now, but the behavior of code can be unexpected in more complicated situations when we want to have multiple variables use the data we've allocated on the heap. Let's explore some of those situations now.

Variables and Data Interacting with Move

Multiple variables can interact with the same data in different ways in Rust. Let's look at an example using an integer in Listing 4-2.

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

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

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

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

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

This looks very similar, so we might assume that the way it works would be the same: that is, the second line would make a copy of the value in `s1` and bind it to `s2`. But this isn't quite what happens.

Take a look at Figure 4-1 to see what is happening to `String` under the covers. A `String` is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

Representation in memory of a `String` holding the value `"hello"` bound to `s1`

The length is how much memory, in bytes, the contents of the `String` are currently using. The capacity is the total amount of memory, in bytes, that the `String` has received from the allocator. The difference between length and capacity matters, but not in this context, so for now, it's fine to ignore the capacity.

When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity stored on the stack. We do not copy the data on the heap that the pointer refers to. The memory looks like this:

Representation after `s2 = s1`

The data pointers point to the same location. This is a problem: when `s2` and `s1` go out of scope, they will both try to free the same memory. This is known as a *double free* error, a type of memory safety bug. Rust prevents this by invalidating `s1` when `s2` takes ownership, so only `s2` will free the memory when it goes out of scope.

Trying to use `s1` after `s2` is created results in a compile-time error:

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

    println!("{s1}, world!"); // Error: borrow of moved value: `s1`
}
```

Error explanation:

> `s1` has been moved to `s2`, so `s1` is no longer valid and cannot be used.

You can clone `s1` if you want a deep copy:

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

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

`clone()` explicitly copies the heap data to create a full, independent copy.

When you see a call to `clone`, it indicates a potentially expensive operation, as it copies data on the heap.

Stack-Only Data: Copy

For simple scalar types like integers, copying is inexpensive and the data is stored completely on the stack. An example:

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

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

In this case, `x` is still valid after copying to `y` because integers implement the `Copy` trait, meaning they are trivially copied instead of moved.

Rust won't allow you to implement `Copy` on types that implement the `Drop` trait, since they need special cleanup.

Typical types that implement `Copy`:

- All integer types like `u32`
- `bool`
- Floating-point types like `f64`
- `char`
- Tuples composed entirely of types that also implement `Copy` (e.g., `(i32, i32)`)

Ownership and Functions

Passing a value to a function either moves or copies it, just like assignment. Example:

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s); // s moves into the function

    let x = 5;
    makes_copy(x); // x is copied, not moved

    println!("{x}"); // Valid since x was copied
}

fn takes_ownership(some_string: String) {
    println!("{some_string}");
} // `some_string` is dropped here, cleaning heap memory

fn makes_copy(some_integer: i32) {
    println!("{some_integer}");
}
```

Using `s` after `takes_ownership(s)` would cause an error because it was moved into the function.

Return Values and Scope

Functions can return ownership of data:

```
fn main() {
    let s1 = gives_ownership(); // s1 takes ownership

    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2); // s2 is moved, s3 takes ownership
}

fn gives_ownership() -> String {
    let some_string = String::from("yours");
    some_string // moved out
}

fn takes_and_gives_back(a_string: String) -> String {
    a_string // moved out
}
```

Similarly, passing and returning ownership is routine, but Rust offers references to avoid transferring ownership.

References

References allow a function to use a value without taking ownership, avoiding the need to transfer or clone data. This is a core feature to work efficiently with data in Rust.

References and Borrowing - The Rust Programming Language

Introduction

The issue with the tuple code in Listing 4-5 is that we have to return the `String` to the calling function so we can still use the `String` after the call to `calculate_length`, because the `String` was moved into `calculate_length`. Instead, we can provide a reference to the `String` value. A *reference* is like a pointer in that it's an address we can follow to access the data stored at that address; that data is owned by some other variable. Unlike a pointer, a reference is guaranteed to point to a valid value of a particular type for the life of that reference.

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

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

    let len = calculate_length(&s1);

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

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

First, notice that all the tuple code in the variable declaration and the function return value is gone. Second, note that we pass `&s1` into `calculate_length` and, in its definition, we take `&String` rather than `String`. These ampersands represent *references*, and they allow you to refer to some value without taking ownership of it. Figure 4-6 depicts this concept.

Figure 4-6: A diagram of `&String s` pointing at `String s1`

Note: The opposite of referencing by using `&` is *dereferencing*, which is accomplished with the dereference operator, `*`. We'll see some uses of the dereference operator in Chapter 8 and discuss details of dereferencing in Chapter 15.

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

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

    let len = calculate_length(&s1);
    println!("The length of '{s1}' is {len}.");
}

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

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

Similarly, the signature of the function uses `&` to indicate that the type of the parameter `s` is a reference. The annotations clarify this:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{s1}' is {len}.");
}
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because s does not have ownership of what it refer
```

The scope in which the variable `s` is valid is the same as any function parameter's scope, but the value pointed to by the reference is not dropped when `s` stops being used, because `s` doesn't have ownership. When functions have references as parameters instead of the actual values, we won't need to return the values in order to give back ownership, because we never had ownership.

We call the action of creating a reference *borrowing*. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back. You don't own it.

Modifying Borrowed Data: Mutable References

What happens if we try to modify something we're borrowing? Try the code in Listing 4-6. Spoiler alert: it doesn't work!

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

    change(&s);
}

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

Error:

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot
  |
help: consider changing this to be a mutable reference
  |
7 | fn change(some_string: &mut String) {
  |                        +++
```

Just as variables are immutable by default, so are references. We're not allowed to modify something we have a reference to.

Mutable References

We can fix the code from Listing 4-6 to allow us to modify a borrowed value with just a few small tweaks that use, instead, a *mutable reference*:

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

    change(&mut s);
}

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

First, we change `s` to be `mut`. Then, we create a mutable reference with `&mut s` where we call the `change` function, and update the function signature to accept a mutable reference with `some_string: &mut String`. This makes it clear that the `change` function will mutate the value it borrows.

Restrictions on Mutable References

Mutable references have one big restriction: if you have a mutable reference to a value, you can have no other references to that value. This code that attempts to create two mutable references to `s` will fail:

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

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

    println!("{}, {}", r1, r2);
}
```

Error:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                         -- first borrow later used here
```

This restriction prevents multiple mutable references at the same time, which allows Rust to prevent data races at compile time. Data races can occur when:

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

Using Scopes to Manage References

You can create a new scope with curly brackets to allow multiple mutable references at different times:

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

```
    {
        let r1 = &mut s;
    } // r1 goes out of scope here, so we can make a new reference with no problems.

    let r2 = &mut s;
}
```

Combining Mutable and Immutable References

Rust enforces similar rules for combining mutable and immutable references. The following code results in an error:

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

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

    println!("{}, {}, and {}", r1, r2, r3);
}
```

Error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
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 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here
```

You cannot have a mutable reference while immutable references exist. Users of an immutable reference don't expect the value to change unexpectedly. Multiple immutable references are allowed, because they only read the data.

Scope and Borrowing

The scope of a reference starts where it is introduced and continues through the last time that reference is used. For example:

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // r1 and r2 are no longer used after this point

    let r3 = &mut s; // no problem
```

```
    println!("{r3}");
}
```

The immutable references `r1` and `r2` end after the `println!`, before the mutable reference `r3` is created. Since their scopes do not overlap, this code is allowed.

Conclusion

Borrowing helps prevent data races and dangling references. Rust enforces rules that:

- You can have either one mutable reference or any number of immutable references at a time.
- References must always be valid.

Next, we will discuss slices.

The Slice Type - The Rust Programming Language

Overview

The Rust Programming Language

The Slice Type

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. A slice is a kind of reference, so it does not have ownership.

Here's a small programming problem: write a function that takes a string of words separated by spaces and returns the first word it finds in that string. If the function doesn't find a space in the string, the whole string must be one word, so the entire string should be returned.

Let's work through how we'd write the signature of this function without using slices, to understand the problem that slices will solve:

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

The `first_word` function has a `&String` as a parameter. We don't need ownership, so this is fine. (In idiomatic Rust, functions do not take ownership of their arguments unless they need to, and the reasons for that will become clear as we keep going!) But what should we return? We don't really have a way to talk about part of a string. However, we could return the index of the end of the word, indicated by a space. Let's try that, as shown in Listing 4-7.

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

```
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()
}
```

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

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

```
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()
}
```

We'll discuss iterators in more detail in Chapter 13. For now, know that `iter` is a method that returns each element in a collection and that `enumerate` wraps the result of `iter` and returns each element as part of a tuple. 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 convenient than calculating the index ourselves.

Because the `enumerate` method returns a tuple, we can use patterns to destructure that tuple. In the `for` loop, we specify a pattern that has `i` for the index in the tuple and `&item` for the single byte in the tuple. Because we get a reference to the element from `.iter().enumerate()`, we use `&` in the pattern.

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

This approach gives us a way to find the index of the end of the first word in the string, but there's a problem. We're returning a `usize` on its own, but it's only a meaningful number in the context of the `&String`. In other words, because it's a separate value from the `String`, there's no guarantee that it will still be valid in the future.

Consider the program in Listing 4-8 that uses the `first_word` function from Listing 4-7.

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

```
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()
}

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 `s` no longer has any content
    // that we could meaningfully use with the value `5`, so `word` is now
    // totally invalid!
}
```

This program compiles without any errors, but the value in `word` becomes invalid after `s.clear()`. Managing such indices manually is tedious and error-prone. Rust provides string slices to address this problem.

String Slices

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

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

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

Rather than a reference to the entire `String`, `hello` is a reference to a portion of the `String`, specified in the extra `[0..5]` part. We create slices using a range within brackets by specifying `[starting_index..ending_index]`.

- `starting_index`: position of the first byte
- `ending_index`: position of one more than the last byte

Internally, the slice data structure stores the starting position and the length of the slice, which corresponds to `ending_index` minus `starting_index`. For example, `let world = &s[6..11];` contains a pointer to the byte at index 6 of `s` with a length of 5.

You can drop the starting or ending index if you want to start from the beginning or go to the end:

```
let slice1 = &s[..2];    // same as &s[0..2]
let slice2 = &s[3..];    // same as &s[3..s.len()]
let slice3 = &s[..];     // the whole string
```

Note: String slice range indices must occur at valid UTF-8 character boundaries; otherwise, a runtime error occurs.

Returning Slices

Let's rewrite `first_word` to return a slice of the string:

```
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[..]
}
```

Now, when we call `first_word`, we get a reference tied to the data, preventing bugs where the string is modified after obtaining a part of it.

Other Slices

Slices are not limited to strings. Arrays can also be sliced:

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

    let slice = &a[1..3];
```

```
    assert_eq!(slice, &[2, 3]);
}
```

This slice has type `&[i32]` and works similarly, referring to part of an array.

Summary

The concepts of ownership, borrowing, and slices ensure memory safety in Rust at compile time. Rust's ownership model manages resources efficiently, preventing many common bugs. This foundation enables writing safe and performant systems programming.

Continue to Chapter 5 for grouping pieces of data in a `struct`.

Defining and Instantiating Structs - The Rust Programming Language

Not Found

Looks like you've taken a wrong turn.

Some things that might be helpful to you though:

Search

§Search

- From the Standard Library
- From DuckDuckGo

Reference

- The Rust official site
- The Rust reference

Docs

The standard library

Rust 1.90.0-nightly
e3843659e

Method Syntax - The Rust Programming Language

Overview

Methods are similar to functions: we declare them with the `fn` keyword and a name, they can have parameters and a return value, and they contain some code that's run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum or a trait object, which we cover in Chapter 6 and Chapter 18, respectively). The first parameter is always `self`, which represents the instance of the struct the method is being called on.

Defining Methods

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

Filename: src/main.rs

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

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

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

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

To define the function within the `Rectangle` context, we start an `impl` block for `Rectangle`. Everything within this block will be associated with the `Rectangle` type. We move the `area` function into this block and change the first parameter to `&self`. In `main`, instead of calling the function with `rect1.area()`, we use method syntax by adding a dot followed by the method name and parentheses.

Note: In the signature for `area`, `&self` is short for `self: &Self`. Within an `impl` block, the type `Self` is an alias for the type the block is for. Methods must have a parameter named `self` of type `Self`. Rust allows abbreviating this with just `self`. The `&` indicates that the method borrows the instance immutably.

Choosing Borrowing or Ownership

Methods can take ownership of `self`, borrow `self` immutably (`&self`), or borrow mutably (`&mut self`). For example, using `&self` allows read-only access, while `&mut self` permits modification.

The main reasons for using methods instead of functions are organization and clarity—grouping capabilities of a type in one `impl` block.

Methods with Same Name as Fields

You can define a method with the same name as a field, for example:

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

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

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

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

When called with parentheses (`rect1.width()`), Rust interprets it as a method; without parentheses (`rect1.width`), it treats it as a field.

The `->` Operator in Methods

In C/C++, you'd use `.` or `->` to call methods, depending on whether you're using an object or a pointer. Rust simplifies this with **automatic referencing and dereferencing**. When calling `object.something()`, Rust adds `&`, `&mut`, or `*` as needed to match the method's signature.

Example:

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn distance(&self, other: &Point) -> f64 {
        let x_squared = f64::powi(other.x - self.x, 2);
```

```
        let y_squared = f64::powi(other.y - self.y, 2);
        f64::sqrt(x_squared + y_squared)
    }
}

let p1 = Point { x: 0.0, y: 0.0 };
let p2 = Point { x: 5.0, y: 6.5 };

p1.distance(&p2);    // calls with automatic referencing
(&p1).distance(&p2); // equivalent
```

Methods with More Parameters

You can add additional parameters after `self`. For example, `can_hold` compares two rectangles:

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

Example usage:

```
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));
}
```

Associated Functions

Functions defined within an `impl` block that don't take `self` are called **associated functions**. They often serve as constructors, e.g., `Rectangle::new()`.

For example, a `square` associated function:

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

fn main() {
    let sq = Rectangle::square(3);
}
```

The `Self` keyword is an alias for the type (`Rectangle`). Call it using the `::` syntax.

Multiple `impl` Blocks

A struct can have multiple `impl` blocks. For example:

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

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
    }
}
```

This is valid and sometimes helpful for organizing code.

Summary

Structs allow creation of custom types that encapsulate meaningful data. In `impl` blocks, you can define functions and methods to specify their behavior. Methods organize functionality related to the type, often with the same name as fields for convenience. Additionally, associated functions act as constructors or utility functions not tied to a specific instance.

Next Topic

Let's explore Rust's enum feature to add more tools to your toolbox.

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Defining an Enum - The Rust Programming Language

The match Control Flow Construct - The Rust Programming Language

The `match` Control Flow Construct

Rust has an extremely powerful control flow construct called `match` that allows you to compare a value against a series of patterns and then execute code based on which pattern matches. Patterns can be made up of literal values, variable names, wildcards, and many other things; Chapter 19 covers all the different kinds of patterns and what they do. The power of `match` comes from the expressiveness of the patterns and the fact that the compiler confirms that all possible cases are handled.

Think of a `match` expression as being like a coin-sorting machine: coins slide down a track with variously sized holes along it, and each coin falls through the first hole it encounters that it fits into. In the same way, values go through each pattern in a `match`, and at the first pattern the value "fits," the value falls into the associated code block to be used during execution.

Speaking of coins, let's use them as an example using `match`! We can write a function that takes an unknown US coin and, in a similar way as the counting machine, determines which coin it is and returns its value in cents, as shown in Listing 6-3.

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

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

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

fn main() {}
```

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

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

When the `match` expression executes, it compares the resultant value against the pattern of each arm, in order. If a pattern matches the value, the code associated with that pattern is executed. If that pattern doesn't

match the value, execution continues to the next arm, much as in a coin-sorting machine. We can have as many arms as we need: in Listing 6-3, our `match` has four arms.

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

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

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

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

fn main() {}
```

Patterns That Bind to Values

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

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

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

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

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

```
}

fn main() {}
```

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

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

```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

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

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

fn main() {
    value_in_cents(Coin::Quarter(UsState::Alaska));
}
```

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

Matching with `Option<T>`

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

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

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

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

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

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

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

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

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

Matches Are Exhaustive

There's one other aspect of `match` we need to discuss: the arms' patterns must cover all possibilities. Consider this version of our `plus_one` function, which has a bug and won't compile:

```
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            Some(i) => Some(i + 1),
        }
    }
    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

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

```
error[E0004]: non-exhaustive patterns: `None` not covered
 --> src/main.rs:3:15
  |
3 |         match x {
```

```
  |               ^ pattern `None` not covered
  |
note: `Option<i32>` defined here
 --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/core/src/option.rs:572:1
 --> /rustc/4eb161250e340c8f48f66e2b929ef4a5bed7c181/library/core/src/option.rs:576:5
  |
  = note: `Option<i32>` pattern is not exhaustively matched
  = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm with a wi
  |
4 |         match x {
5 |             Some(i) => Some(i + 1),
6 |             None => todo!(),
7 |         }
```

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

Catch-All Patterns and the `_` Placeholder

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

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

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

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

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

Rust also has a pattern we can use when we want a catch-all but don't want to *use* the value in the catch-all pattern: `_` is a special pattern that matches any value and does not bind to that value. This tells Rust we aren't going to use the value, so Rust won't warn us about an unused variable.

Let's change the rules of the game: now, if you roll anything other than a 3 or 7, you must roll again. We no longer need to use the catch-all value, so we can change our code to use `_` instead of the variable named `other`:

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

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

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

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

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

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

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

There's more about patterns and matching that we'll cover in Chapter 19. For now, we're going to move on to the `if let` syntax, which can be useful in situations where the `match` expression is a bit wordy.

Concise Control Flow with `if let` and `let else` - The Rust Programming Language

Packages and Crates - The Rust Programming Language

Overview

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

A *crate* is the smallest amount of code that the Rust compiler considers at a time. Even if you run `rustc` rather than `cargo` and pass a single source code file (as we did all the way back in "Writing and Running a Rust Program" in Chapter 1), the compiler considers that file to be a crate. Crates can contain modules, and the modules may be defined in other files that get compiled with the crate, as we'll see in the coming sections.

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

Library crates don't have a `main` function, and they don't compile to an executable. Instead, they define functionality intended to be shared with multiple projects. For example, the `rand` crate we used in Chapter 2 provides functionality that generates random numbers.
Most of the time when Rustaceans say "crate," they mean **library crate**, and they use "crate" interchangeably with the general programming concept of a "library."

Crate Root

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

Packages

A *package* is a bundle of one or more crates that provides a set of functionality.
A package contains a `Cargo.toml` file that describes how to build those crates. Cargo is actually a package that contains the binary crate for the command line tool you've been using to build your code. The package also contains a library crate that the binary crate depends on. Other projects can depend on the Cargo library crate to use the same logic the Cargo command line tool uses.

A package can contain as many binary crates as you like, but at most only one library crate.
A package must contain at least one crate, whether that's a library or binary crate.

Creating a Package

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

```
cargo new my-project
```

which outputs:

```
Created binary (application) `my-project` package
```

In the project directory, there's a `Cargo.toml` file, giving us a package.
There's also an `src` directory that contains `main.rs`. Open `Cargo.toml` in your text editor, and note there's no mention of `src/main.rs`. Cargo follows a convention that `src/main.rs` is the crate root of a

binary crate with the same name as the package.
Likewise, Cargo knows that if the package directory contains `src/lib.rs`, the package contains a library crate with the same name as the package, and `src/lib.rs` is its crate root. Cargo passes the crate root files to `rustc` to build the library or binary.

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

Not Found

Looks like you've taken a wrong turn.

Some things that might be helpful to you though:

§ Search

- From the Standard Library
- From DuckDuckGo

§ Reference

- The Rust official site
- The Rust reference

§ Docs

- The standard library

Rust 1.90.0-nightly
e3843659e

Paths for Referring to an Item in the Module Tree - The Rust Programming Language

---

Overview

To show Rust where to find an item in a module tree, we use a path similarly to navigating a filesystem. To call a function, we need its path.

Types of Paths

- **Absolute path**: The full path starting from a crate root; for external crates, begins with the crate name, and for the current crate, starts with `crate`.
- **Relative path:** Starts from the current module, using `self`, `super`, or an identifier within the current module.

Both follow with one or more identifiers separated by `::`.

Example: Calling `add_to_waitlist`

Listing 7-3 demonstrates calling `add_to_waitlist` using absolute and relative paths.

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

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

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

Explanation:

- Absolute path starts with `crate` and drills down to `add_to_waitlist`.
- Relative path starts within the current module, referencing `front_of_house`.

Access Restrictions and Privacy Rules

Attempting to compile the above results in errors:

```
error[E0603]: module `hosting` is private
...
```

```
note: the module `hosting` is defined here
```

- Modules and items are private by default.
- To access items in child modules, the parent module must declare them as `pub`.

Making Modules and Items Public

Declare modules as public

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

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

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

Error persists because functions are still private

To fix this, declare the function as `pub`:

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

Now, the function can be called outside its module.

Module and Function Visibility

- Declaring a module `pub` makes the module accessible from outside.
- Declaring a function `pub` within a `pub` module makes it accessible.
- `pub` on a module does **not** automatically make its contents public; each item must be explicitly made public.

Using `pub` to Expose Paths

Example:

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

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

Result: Both paths work due to `pub` declarations.

Absolute vs. Relative Paths

- **Absolute path**: Starts from `crate`, the root of the module tree.
- **Relative path**: Begins within the current module, using `front_of_house`.

Key points:

- If the modules are `pub`, you can access them from outside.
- `pub fn` allows functions to be called from other modules.
- Making modules or functions private restricts access.

Privacy Details

- Items are private by default.
- `pub` makes modules, functions, structs, or enums accessible from outside.
- Items within a module are private unless declared `pub`.

Additional Example: `pub` on Functions and Structs

Making a function public:

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

Making a struct `pub`, but with private fields:

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

- `toast` can be accessed and modified externally.
- `seasonal_fruit` remains private.

Making enum variants public:

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

- All variants are automatically public if enum is `pub`.

Summary

- Use `pub` to expose modules, functions, structs, and enum variants.
- Items are private by default.
- Proper use of `pub` and module organization helps control access and encapsulation.

Next Topic

- **The `use` keyword** for bringing paths into scope.
- Combining `pub` with `use` to organize public APIs.

Not Found

Looks like you've taken a wrong turn.

Some things that might be helpful to you though:

Search

From the Standard Library From DuckDuckGo

Reference

- The Rust official site
- The Rust reference

Docs

The standard library

Separating Modules into Different Files - The Rust Programming Language

---

[Content omitted for brevity; the rest of the document continues with detailed explanations about Rust modules and file organization.]

Summary

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

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