Getting Started

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

Installing Rust on Linux, macOS, and Windows

Writing a program that prints Hello, world!

Using cargo, Rust’s package manager and build system

Installation

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

Note If you prefer not to use rustup for some reason, please see the Other Rust Installation Methods page at https://forge.rust-lang.org/infra/other-installation-methods.html for more options.

The following steps install the latest stable version of the Rust compiler. Rust’s stability guarantees ensure that all the examples in the book that compile will continue to compile with newer Rust versions. The output might differ slightly between versions because Rust often improves error messages and warnings. In other words, any newer, stable version of Rust you install using these steps should work as expected with the content of this book.

box

Command Line Notation

In this chapter and throughout the book, we’ll show some commands used in the terminal. Lines that you should enter in a terminal all start with $. You don’t need to type the $ character; it’s the command line prompt shown to indicate the start of each command. Lines that don’t start with $ typically show the output of the previous command. Additionally, PowerShell-specific examples will use > rather than $.

Installing rustup on Linux or macOS

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

$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh

The command downloads a script and starts the installation of the rustup tool, which installs the latest stable version of Rust. You might be prompted for your password. If the install is successful, the following line will appear:

Rust is installed now. Great!

You will also need a linker, which is a program that Rust uses to join its compiled outputs into one file. It is likely you already have one. If you get linker errors, you should install a C compiler, which will typically include a linker. A C compiler is also useful because some common Rust packages depend on C code and will need a C compiler.

On macOS, you can get a C compiler by running:

$ xcode-select --install

Linux users should generally install GCC or Clang, according to their distribution’s documentation. For example, if you use Ubuntu, you can install the build-essential package.

Installing rustup on Windows

On Windows, go to https://www.rust-lang.org/tools/install and follow the instructions for installing Rust. At some point in the installation, you’ll be prompted to install Visual Studio. This provides a linker and the native libraries needed to compile programs. If you need more help with this step, see https://rust-lang.github.io/rustup/installation/windows-msvc.html.

The rest of this book uses commands that work in both cmd.exe and PowerShell. If there are specific differences, we’ll explain which to use.

Troubleshooting

To check whether you have Rust installed correctly, open a shell and enter this line:

$ rustc --version

You should see the version number, commit hash, and commit date for the latest stable version that has been released, in the following format:

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

If you see this information, you have installed Rust successfully! If you don’t see this information, check that Rust is in your %PATH% system variable as follows.

In Windows CMD, use:

> echo %PATH%

In PowerShell, use:

> echo $env:Path

In Linux and macOS, use:

$ echo $PATH

If that’s all correct and Rust still isn’t working, there are a number of places you can get help. Find out how to get in touch with other Rustaceans (a silly nickname we call ourselves) on the community page at https://www.rust-lang.org/community.

Updating and Uninstalling

Once Rust is installed via rustup, updating to a newly released version is easy. From your shell, run the following update script:

$ rustup update

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

$ rustup self uninstall

Reading the Local Documentation

The installation of Rust also includes a local copy of the documentation so that you can read it offline. Run rustup doc to open the local documentation in your browser.

Any time a type or function is provided by the standard library and you’re not sure what it does or how to use it, use the application programming interface (API) documentation to find out!

Using Text Editors and IDEs

This book makes no assumptions about what tools you use to author Rust code. Just about any text editor will get the job done! However, many text editors and integrated development environments (IDEs) have built-in support for Rust. You can always find a current list of many editors and IDEs on the tools page at https://www.rust-lang.org/tools on the Rust website.

Hello, World!

Now that you’ve installed Rust, it’s time to write your first Rust program. It’s traditional when learning a new language to write a little program that prints the text Hello, world! to the screen, so we’ll do the same here!

Note This book assumes basic familiarity with the command line. Rust makes no specific demands about your editing or tooling or where your code lives, so if you prefer to use an IDE instead of the command line, feel free to use your favorite IDE. Many IDEs now have some degree of Rust support; check the IDE’s documentation for details. The Rust team has been focusing on enabling great IDE support via rust-analyzer. See Appendix D for more details.

Project Directory Setup

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

Open a terminal and enter the following commands to make a projects directory and a directory for the “Hello, world!” project within the projects directory.

For Linux, macOS, and PowerShell on Windows, enter this:

$ mkdir ~/projects

$ cd ~/projects

$ mkdir hello\_world

$ cd hello\_world

For Windows CMD, enter this:

> mkdir "%USERPROFILE%\projects"

> cd /d "%USERPROFILE%\projects"

> mkdir hello\_world

> cd hello\_world

Rust Program Basics

Next, make a new source file and call it main.rs. Rust files always end with the .rs extension. If you’re using more than one word in your filename, the convention is to use an underscore to separate them. For example, use hello\_world.rs rather than helloworld.rs.

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

main.rs

fn main() {

println!("Hello, world!");

}

A program that prints Hello, world!

Save the file and go back to your terminal window in the ~/projects/hello\_world directory. On Linux or macOS, enter the following commands to compile and run the file:

$ rustc main.rs

$ ./main

Hello, world!

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

> rustc main.rs

> .\main

Hello, world!

Regardless of your operating system, the string Hello, world! should print to the terminal. If you don’t see this output, refer back to “Troubleshooting” on page XX for ways to get help.

If Hello, world! did print, congratulations! You’ve officially written a Rust program. That makes you a Rust programmer—welcome!

Anatomy of a Rust Program

Let’s review this “Hello, world!” program in detail. Here’s the first piece of the puzzle:

fn main() {

}

These lines define a function named main. The main function is special: It is always the first code that runs in every executable Rust program. Here, the first line declares a function named main that has no parameters and returns nothing. If there were parameters, they would go inside the parentheses (()).

The function body is wrapped in curly brackets ({}). Rust requires curly brackets around all function bodies. It’s good style to place the opening curly bracket on the same line as the function declaration, adding one space in between.

Note If you want to stick to a standard style across Rust projects, you can use an automatic formatter tool called rustfmt to format your code in a particular style (more on rustfmt in Appendix D). The Rust team has included this tool with the standard Rust distribution, as rustc is, so it should already be installed on your computer!

The body of the main function holds the following code:

println!("Hello, world!");

This line does all the work in this little program: It prints text to the screen. There are three important details to notice here.

First, println! calls a Rust macro. If it had called a function instead, it would be entered as println (without the !). Rust macros are a way to write code that generates code to extend Rust syntax, and we’ll discuss them in more detail in Chapter 20. For now, you just need to know that using a ! means that you’re calling a macro instead of a normal function and that macros don’t always follow the same rules as functions.

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

Third, we end the line with a semicolon (;), which indicates that this expression is over, and the next one is ready to begin. Most lines of Rust code end with a semicolon.

Compilation and Execution

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

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

$ rustc main.rs

If you have a C or C++ background, you’ll notice that this is similar to gcc or clang. After compiling successfully, Rust outputs a binary executable.

On Linux, macOS, and PowerShell on Windows, you can see the executable by entering the ls command in your shell:

$ ls

main main.rs

On Linux and macOS, you’ll see two files. With PowerShell on Windows, you’ll see the same three files that you would see using CMD. With CMD on Windows, you would enter the following:

> dir /B %= the /B option says to only show the file names =%

main.exe

main.pdb

main.rs

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

$ ./main # or .\main on Windows

If your main.rs is your “Hello, world!” program, this line prints Hello, world! to your terminal.

If you’re more familiar with a dynamic language, such as Ruby, Python, or JavaScript, you might not be used to compiling and running a program as separate steps. Rust is an ahead-of-time compiled language, meaning you can compile a program and give the executable to someone else, and they can run it even without having Rust installed. If you give someone a .rb, .py, or .js file, they need to have a Ruby, Python, or JavaScript implementation installed (respectively). But in those languages, you only need one command to compile and run your program. Everything is a trade-off in language design.

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

Hello, Cargo!

Cargo is Rust’s build system and package manager. Most Rustaceans use this tool 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 “Installation” on page XX. 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.

Cargo.toml

[package]

name = "hello\_cargo"

version = "0.1.0"

edition = "2024"

[dependencies]

Contents of Cargo.toml generated by cargo new

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  , so we’ll use this dependencies section then.

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

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.

Leveraging Cargo’s Conventions

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. In fact, 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 at https://doc.rust-lang.org/cargo.

Summary

You’re already off to a great start on your Rust journey! In this chapter, you 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.