[Installation 1](#_Toc510079705)

[Installing Rustup on Linux or macOS 2](#_Toc510079706)

[Installing Rustup on Windows 3](#_Toc510079707)

[Custom Installations Without Rustup 3](#_Toc510079708)

[Updating and Uninstalling 3](#_Toc510079709)

[Troubleshooting 3](#_Toc510079710)

[Local Documentation 4](#_Toc510079711)

[Hello, World! 4](#_Toc510079712)

[Creating a Project Directory 4](#_Toc510079713)

[Writing and Running a Rust Program 5](#_Toc510079714)

[Anatomy of a Rust Program 6](#_Toc510079715)

[Compiling and Running Are Separate Steps 7](#_Toc510079716)

[Hello, Cargo! 8](#_Toc510079717)

[Creating a Project with Cargo 9](#_Toc510079718)

[Building and Running a Cargo Project 11](#_Toc510079719)

[Building for Release 12](#_Toc510079720)

[Cargo as Convention 13](#_Toc510079721)

[Summary 13](#_Toc510079722)

1

Getting Started

prod: Let’s try setting the bullet points in the same size font as 1st para, and see how that looks.

Let’s start your Rust journey! 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.

The following steps install the latest stable version of the Rust compiler. All the examples and output in this book use stable Rust 1.21.0. 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.

Start 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 in the $ character; it simply indicates the start of each command. Many tutorials use the convention $ for commands you run as a regular user and # for commands you run as an administrator. Lines that don’t start with $  typically show the output of the previous command. Additionally, PowerShell specific examples will use > rather than $.

End box

Installing Rustup on Linux or macOS

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

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

The command downloads a script and starts the installation of the rustup tool, 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!

Of course, if you distrust using curl URL | sh to install software, you can download, inspect, and run the script however you like.

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

$ source $HOME/.cargo/env

Alternatively, you can add the following line to your ~/.bash\_profile:

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

Additionally, you’ll need a linker of some kind. It’s likely one is already installed, but when you try to compile a Rust program and get errors indicating that a linker could not execute, you’ll need to install one. You can install a C compiler, because that will usually come with the correct linker. Check your platform’s documentation for how to install a C compiler. Some common Rust packages depend on C code and will need a C compiler too, so it might be worth installing one now regardless.

Installing Rustup on Windows

On Windows, go to https://www.rust-lang.org/en-US/install.html and follow the instructions for installing Rust. At some point in the installation, you’ll receive a message explaining that you’ll also need the C++ build tools for Visual Studio 2013 or later. The easiest way to acquire the build tools is to install Build Tools for Visual Studio 2017 at https://www.visualstudio.com/downloads/. The tools are in the Other Tools and Frameworks section.

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.

Custom Installations Without Rustup

If you prefer not to use rustup for some reason, please see the Rust installation page at https://www.rust-lang.org/install.html for other options.

Updating and Uninstalling

After you’ve installed Rust via rustup, updating to the latest version is easy. From your shell, run the following update script:

$ rustup update

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

$ rustup self uninstall

Troubleshooting

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

$ rustc --version

At the time you install, you should see the version number, commit hash, and commit date for the latest stable version in the following format:

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

If you see this information, Rust has installed successfully! If you don’t see this information and you’re on Windows, check that Rust is in your %PATH% system variable. If that’s all correct and Rust still isn’t working, there are a number of places you can get help. The easiest is the #rust IRC channel on irc.mozilla.org, which you can access through Mibbit at http://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust/. At that address you can chat with other Rustaceans (a silly nickname we call ourselves) who can help you out. Other great resources include the Users forum at https://users.rust-lang.org/ and Stack Overflow at http://stackoverflow.com/questions/tagged/rust/.

Local Documentation

The installer also includes a copy of the documentation locally, so 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!

Hello, World!

Now that you’ve installed Rust, let’s 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, tooling, or where your code lives, so if you prefer to use an integrated development environment (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. Recently, the Rust team has been focusing on enabling great IDE support, and progress has been made rapidly on that front!

Creating a Project Directory

You’ll start by making a directory to store your Rust code. It doesn’t matter to Rust where your code lives, but for the exercises and projects in this book, you’ll make a projects directory in your home directory to keep 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 and macOS, enter this:

$ mkdir ~/projects

$ cd ~/projects

$ mkdir hello\_world

$ cd hello\_world

For Windows CMD, enter this:

> mkdir "%USERPROFILE%\projects"

> cd /d "%USERPROFILE%\projects"

> mkdir hello\_world

> cd hello\_world

For Windows PowerShell, enter this:

> mkdir $env:USERPROFILE\projects

> cd $env:USERPROFILE\projects

> mkdir hello\_world

> cd hello\_world

Writing and Running a Rust Program

Next, make a new source file and call it main.rs. Rust files always end with the .rs extension. If you’re using more than one word in your filename, 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!");

}

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

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

$ rustc main.rs

$ ./main

Hello, world!

On Windows, enter .\main.exe instead of ./main.

> rustc main.rs

> .\main.exe

Hello, world!

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

prod: link xref (this chapter)

If you did see Hello, world! printed, congratulations! You’ve officially written a Rust program. That makes you a Rust programmer! Welcome!

Anatomy of a Rust Program

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

fn main() {

}

These lines define a function in Rust. The main function is special: it is always the first code that runs in every executable Rust program. 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, ( and ).

Also, note that the function body is wrapped in curly brackets, { and }. Rust requires these 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.

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

Inside the main function is the following code:

println!("Hello, world!");

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

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

prod: check xref

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

Fourth, 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.

Compiling and Running Are Separate Steps

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

Before running a Rust program, you must compile it using the Rust compiler 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 as follows:

$ ls

main main.rs

With CMD on Windows, you would enter the following:

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

main.exe

main.pdb

main.rs

This shows two files: the source code with the .rs extension and the executable (main.exe on Windows, but main on all other platforms). From here, you just run the main or main.exe file, like this:

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

If main.rs was your “Hello, world!” program, this line would print Hello, world! 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, 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 libraries your code needs dependencies.)

The simplest Rust programs, like the one we’ve written so far, don’t have any dependencies. So 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 on page XX. If you installed Rust through some other means, check whether Cargo is installed by entering the following into your terminal:

prod: check & fill xref (this chapter)

$ 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 --bin

$ cd hello\_cargo

This code creates a new binary executable called hello\_cargo. The --bin argument passed to cargo new makes an executable application (often just called a binary) as opposed to a library. 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.

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"

authors = ["Your Name <you@example.com>"]

[dependencies]

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

This file is in the TOML (Tom’s Obvious, Minimal Language) format, which is 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 who wrote it. Cargo gets your name and email information from your environment, so if that information is not correct, fix it now and then save the file.

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.

prod: confirm xref

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 previous project and the project Cargo generates are that Cargo’s code is stored 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. So 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 our project in the hello\_world directory, you can convert it to a project that does use Cargo. Just move the project code into the src directory and create an appropriate Cargo.toml file.

Building and Running a Cargo Project

Now let’s look at the difference when we build and run the “Hello, world!” program in Cargo! From your project directory, build your project by entering the following commands:

$ cargo build

Compiling hello\_cargo v0.1.0 (file:///projects/hello\_cargo)

Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs

This code creates an executable file in target/debug/hello\_cargo (or target\debug\hello\_cargo.exe on Windows), which you can run 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 creates Cargo.lock, which 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 and then run code all in one step:

$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/hello\_cargo`

Hello, world!

Notice that this time we didn’t see output indicating that Cargo was compiling hello\_cargo. Cargo figured out that the files hadn’t changed, so it 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 uses cargo check. This command quickly checks your code to make sure it compiles but doesn’t produce an executable:

$ cargo check

Compiling hello\_cargo v0.1.0 (file:///projects/hello\_cargo)

Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs

Why would you not want an executable? Often, cargo check is much faster 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! 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 give it a spin themselves.

To recap, the benefits of using Cargo are:

We can build a project using cargo build or cargo check.

We can build and run a project in one step using cargo run.

Instead of the result of the build being saved 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. With complex projects composed of multiple crates, 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 the project directory, and build:

$ git clone someurl.com/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’ve learned how to:

Install the latest stable version of Rust

Write a “Hello, world!” program using rustc directly and 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 the next chapter, 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.