

The Rust Programming Language

by Steve Klabnik and Carol Nichols, with contributions from the Rust Community

This version of the text assumes you're using Rust 1.67.1 (released 2023-02-09) or later. See the ["Installation" section of Chapter 1](#) to install or update Rust.

The HTML format is available online at <https://doc.rust-lang.org/stable/book/> and offline with installations of Rust made with `rustup`; run `rustup docs --book` to open.

Several community [translations](#) are also available.

This text is available in [paperback and ebook format](#) from No Starch Press.



Want a more interactive learning experience? Try out a different version of the Rust Book, featuring: quizzes, highlighting, visualizations, and more:

<https://rust-book.cs.brown.edu>

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

Introduction

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

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

Who Rust Is For

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

Teams of Developers

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

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

- Cargo, the included dependency manager and build tool, makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem.
- The Rustfmt formatting tool ensures a consistent coding style across developers.
- The Rust Language Server powers Integrated Development Environment (IDE) integration for code completion and inline error messages.

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

Students

Rust is for students and those who are interested in learning about systems concepts. Using Rust, many people have learned about topics like operating systems development. The community is very welcoming and happy to answer student questions. Through efforts such as this book, the Rust teams want to make systems concepts more accessible to more people, especially those new to programming.

Companies

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

Open Source Developers

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

People Who Value Speed and Stability

Rust is for people who crave speed and stability in a language. By speed, we mean both how quickly Rust code can run and the speed at which Rust lets you write programs. The Rust compiler's checks ensure stability through feature additions and refactoring. This is in contrast to the brittle legacy code in languages without these checks, which developers are often afraid to modify. By striving for zero-cost abstractions, higher-level features that compile to lower-level code as fast as code written manually, Rust endeavors to make safe code be fast code as well.

The Rust language hopes to support many other users as well; those mentioned here are merely some of the biggest stakeholders. Overall, Rust's greatest ambition is to eliminate the trade-offs that programmers have accepted for decades by providing *safety and productivity, speed and ergonomics*. Give Rust a try and see if its choices work for you.

Who This Book Is For

This book assumes that you've written code in another programming language but

doesn't make any assumptions about which one. We've tried to make the material broadly accessible to those from a wide variety of programming backgrounds. We don't spend a lot of time talking about what programming *is* or how to think about it. If you're entirely new to programming, you would be better served by reading a book that specifically provides an introduction to programming.

How to Use This Book

In general, this book assumes that you're reading it in sequence from front to back. Later chapters build on concepts in earlier chapters, and earlier chapters might not delve into details on a particular topic but will revisit the topic in a later chapter.

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

Chapter 1 explains how to install Rust, how to write a "Hello, world!" program, and how to use Cargo, Rust's package manager and build tool. Chapter 2 is a hands-on introduction to writing a program in Rust, having you build up a number guessing game. Here we cover concepts at a high level, and later chapters will provide additional detail. If you want to get your hands dirty right away, Chapter 2 is the place for that. Chapter 3 covers Rust features that are similar to those of other programming languages, and in Chapter 4 you'll learn about Rust's ownership system. If you're a particularly meticulous learner who prefers to learn every detail before moving on to the next, you might want to skip Chapter 2 and go straight to Chapter 3, returning to Chapter 2 when you'd like to work on a project applying the details you've learned.

Chapter 5 discusses structs and methods, and Chapter 6 covers enums, `match` expressions, and the `if let` control flow construct. You'll use structs and enums to make custom types in Rust.

In Chapter 7, you'll learn about Rust's module system and about privacy rules for organizing your code and its public Application Programming Interface (API). Chapter 8 discusses some common collection data structures that the standard library provides, such as vectors, strings, and hash maps. Chapter 9 explores Rust's error-handling philosophy and techniques.

Chapter 10 digs into generics, traits, and lifetimes, which give you the power to define code that applies to multiple types. Chapter 11 is all about testing, which even with Rust's safety guarantees is necessary to ensure your program's logic is correct. In Chapter 12, we'll build our own implementation of a subset of functionality from the `grep` command line tool that searches for text within files. For this, we'll use many of the concepts we

discussed in the previous chapters.

Chapter 13 explores closures and iterators: features of Rust that come from functional programming languages. In Chapter 14, we'll examine Cargo in more depth and talk about best practices for sharing your libraries with others. Chapter 15 discusses smart pointers that the standard library provides and the traits that enable their functionality.

In Chapter 16, we'll walk through different models of concurrent programming and talk about how Rust helps you to program in multiple threads fearlessly. Chapter 17 looks at how Rust idioms compare to object-oriented programming principles you might be familiar with.


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



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

Finally, some appendices contain useful information about the language in a more reference-like format. Appendix A covers Rust's keywords, Appendix B covers Rust's operators and symbols, Appendix C covers derivable traits provided by the standard library, Appendix D covers some useful development tools, and Appendix E explains Rust editions. In Appendix F, you can find translations of the book, and in Appendix G we'll cover how Rust is made and what nightly Rust is.

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

An important part of the process of learning Rust is learning how to read the error messages the compiler displays: these will guide you toward working code. As such, we'll provide many examples that don't compile along with the error message the compiler will show you in each situation. Know that if you enter and run a random example, it may not compile! Make sure you read the surrounding text to see whether the example you're trying to run is meant to error. Ferris will also help you distinguish code that isn't meant to work:

Ferris	Meaning
	This code does not compile!

Ferris	Meaning
	This code panics!
	This code does not produce the desired behavior.

In most situations, we'll lead you to the correct version of any code that doesn't compile.

Source Code

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

Getting Started

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

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

Installation

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

Note: If you prefer not to use `rustup` for some reason, please see the [Other Rust Installation Methods](#) page 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.

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 receive a message explaining that you'll also need the MSVC build tools for Visual Studio 2013 or later.

To acquire the build tools, you'll need to install [Visual Studio 2022](#). When asked which workloads to install, include:

- "Desktop Development with C++"
- The Windows 10 or 11 SDK
- The English language pack component, along with any other language pack of your choosing

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](#).

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
```

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!

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 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. The Rust team has been focusing on enabling great IDE support via `rust-analyzer`. See [Appendix D](#) for more details.

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, 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
```

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, 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.

Filename: main.rs

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

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

Save the file and go back to your terminal window 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.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 the [“Troubleshooting”](#) part of the Installation section 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 `{}`. 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 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 had called a function instead, it would be entered as `println` (without the `!`). We'll discuss Rust macros in more detail in Chapter 19. 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.

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:

```
$ 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.exe 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 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 = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[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 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.

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. 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](#).

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.