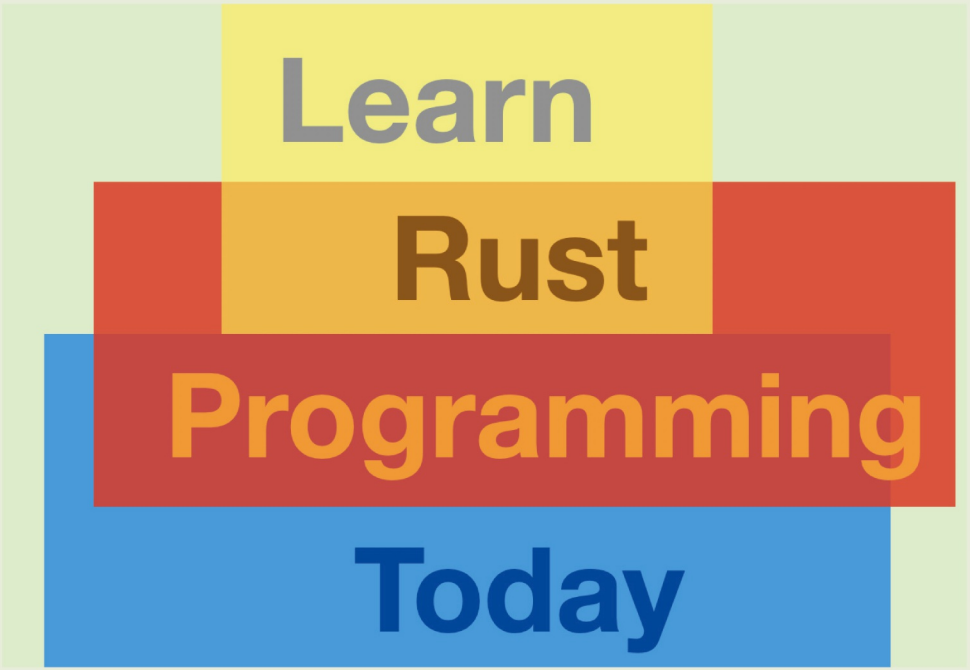


JERE KÄPYAHO

Learn Rust Programming Today

Learn Rust Programming Today

From Building Blocks to a Working Application

Jere Käpyaho

Editing and cover design: Conifer Productions Oy, Tampere, Finland

Kustantaja: BoD · Books on Demand GmbH, Helsinki, Suomi

Valmistaja: BoD · Books on Demand GmbH, Überseering 33, 22297 Hamburg, Saksa

ISBN: 978-952-89-5090-5

Table of Contents

List of Examples

Introduction

Ten years after the release of version 1.0, Rust has become a force to be reckoned with. When advocating Rust, the usual statistics and anecdotes are rolled out:

- Rust has nine consecutive years as the most loved / admired programming language in the Stack Overflow annual survey.
- As this book is being completed Rust has reached an all-time high position in the TIOBE Index[1] (June 2026, 12th).
- Rust is now core part of the Linux kernel[2] (not without controversy, of course).
- Microsoft is reducing the attack surface of shipping products using Rust or lessons learned from it[3].
- Amazon Web Services has developed the Bottlerocket container operating system mostly in Rust[4].
- AdaCore is diversifying from Ada and SPARK to include Rust[5].

This kind of reporting really belongs in InfoWorld, Ars Technica or The Register instead of a Rust programming book, but it cannot be denied that there is momentum.

Before we head straight into practical Rust programming, let's have a quick look at how Rust can be compared with other languages that might be more familiar to the reader.

[1] https://www.tiobe.com/tiobe-index/
[2] https://rust-for-linux.com
[3] https://techcommunity.microsoft.com/blog/surfaceitpro/safer-drivers-stronger-devices/4431411
[4] https://bottlerocket.dev
[5] https://www.adacore.com/languages/rust

Rust in a nutshell

If you already have some (or a lot of) programming experience, you have come across many features that somehow define programming languages. It can be useful to frame Rust in terms of those features.

Here are some of the most obvious differences between Rust and other current programming languages. You can skip this section if you want to get straight to the action.

Notational differences

While Rust looks at least a little familiar to those coming from "curly brackets" languages, it's good to know about some notational differences that can make your work a little easier.

- *Curly brackets:* Like C, C++, Java, JavaScript, Swift, and many others, Rust uses curly brackets to delimit blocks. However, unlike Python, whitespace is not significant.
- *Less parentheses:* Rust does not need parentheses around expressions in conditional statements. They can be used to provide clarity or force an evaluation order, but the traditional outer parentheses are not needed.
- *Semicolons for statements only:* The semicolon is used to terminate a statement, but not after an expression. This has an interesting application in functions, where the last expression in the function body becomes the return value.
- *Commas allowed:* Commas are allowed after the last item in arrays and other list-like constructs. This is to avoid a compiler error when you add an item to the end of the list, but forget to add the comma.
- *Terseness:* Rust uses short keywords and symbols for common semantic elements, such as `fn`, `struct`, and `impl` instead of `function`, `record`, or `implementation`.
- *Pattern matching:* Rust has no traditional `switch` statement. Instead the `match` expression uses pattern matching to select between alternatives.

- *No function overloading:* Functions with the same name but different argument lists are not allowed in Rust.
- *No default arguments in functions*, and no named arguments either.
- *Explicit block structure:* curly brackets are always needed in if expressions and loops.

Conceptual differences

- *Errors, not exceptions:* Rust does not have exceptions like C++ or Java, so there is no `try...catch` construct either. Errors are handled using a dedicated `Result` generic enumeration type with `Ok` and `Err` variants.
- *No* `null`*:* Rust does not have the `null` value, so the same kind of null pointer exceptions that are common in Java can't occur. Instead Rust has the `Option` type which can be inspected.
- *Stack instead of heap:* Memory for values in the program is usually allocated on the stack, unless dynamic allocation on the heap is specifically asked for.
- *No garbage collection:* The Rust compiler checks the program for memory usage. Allocated memory is freed when the variables go out of scope. The compiler detects cases where memory would be used after it has been freed.
- *Powerful macro system:* Rust macros are different from traditional C-language macros that do text substitution. They generate code when they are expanded early in the compilation process.

Learning Rust by doing

There is no getting around the fact that Rust can be a difficult language to learn. It has what is known as a "steep learning curve". Even though it is an imperative language that at least superficially looks a bit like the "curly bracket languages"—C, C++, Java, C#, JavaScript—there are many concepts that are very different from those more established languages.

It is also somewhat challenging to find a linear ordering for all the features of Rust that matter in everyday software development. When learning Rust

new things come up all the time, and you will either have to deal with them on the spot, or defer them to some future time. Both of these approaches have their problems.

In this book I've decided to approach Rust programming from the viewpoint of learning by doing, instead of isolated examples that you "just have to remember" or "keep in mind". Apart from the very first small examples of Rust programs, almost everything in this book is aimed towards a single end result: a working application that does a well-defined thing.

The Today application is large enough to provide an opportunity to meet most, if not all, the important concepts of Rust in action, and to show a way to take advantage of them in the context of real software development.

It is my hope that by the time you finish this book you have a much better idea about what Rust is good for, and how to use that goodness to your advantage. You will also have a useful, flexible application that can serve as the basis for further development or as a source of building blocks for other applications.

Who this book is for

This book is intended primarily for programmers who already have a good command of one or more programming languages, and who are approaching Rust as a possible "second language", maybe even with thoughts of making it the first. This book will probably not work for you if you are a complete beginner, but go ahead and pick it up anyway, out of curiosity. You can fill in the gaps with other books in the Rust lexicon, like *The Rust Programming Language* [Kla2026].

If you have ever felt frustrated with programming books that present seemingly disconnected facts about language syntax but never a coherent picture of what can be achieved with the concepts of the language, this could be the book for you. Instead of isolated examples this book has a long arc, where even the first small ideas are somehow connected to the end result (read on and you shall see).

This book is primarily about Rust the language, but it is also about Rust the enabler. Rust is one of the few current languages where everything comes to-

gether nicely, once you have given it a little bit of time and practice. The language is coherent, the tools are excellent, and the results are great.

Source code repositories

All the programs developed in this book are available on GitHub under the MIT License. The main repository[6] has all the source code for every chapter, in a format that should get you up and running with just `cargo run` (see Chapter 5, *Getting to know Cargo* for details).

The book repository has a chapter key that connects the chapter numbers to the directories in the repository. See the `README.md` in the repository for details.

The Today program is developed separately, starting from where this book ends, in its own repository[7]. It is anticipated that this repository will soon leave the book repository behind in terms of features.

Acknowledgments

Thanks to all the numerous open source developers who made the tools that enabled this book: the Rust compiler and tools, Visual Studio Code, Emacs, Vim, DocBook XML, DocBook XSL stylesheets, Apache FOP, SQLite, xmllint, Darwin, Linux, the utilities we use daily, and so much more. Thanks also to all the closed-source authors who write software that manages traffic, finance, municipalities, vehicles, and entertainment.

Thanks to the friends, colleagues, and students who listened politely to my enthusiastic rants about Rust and this book.

Thanks to the music makers who provided the soundtrack to writing this book, including but not limited to: Beach House, Carbon Based Lifeforms, Yellow Magic Orchestra, Hiroshi Yoshimura, Rudy Adrian, and many more.

And as always, a hug/e thank you to my love Virpi for the support and encouragement.

[6] https://github.com/coniferprod/learn-rust
[7] https://github.com/coniferprod/today-rs

Part I. Getting Started with Rust

In this part of the book we will look at installing the Rust tools, and then go for a test drive at the Rust Playground. We'll also get to know how to invoke the Rust compiler to turn source code into an executable file, and take a look at how to edit Rust source code with a programmer's editor. Finally, we'll introduce Cargo, the build system and package manager of Rust.

Table of Contents

Chapter 1. Using the Rust Playground

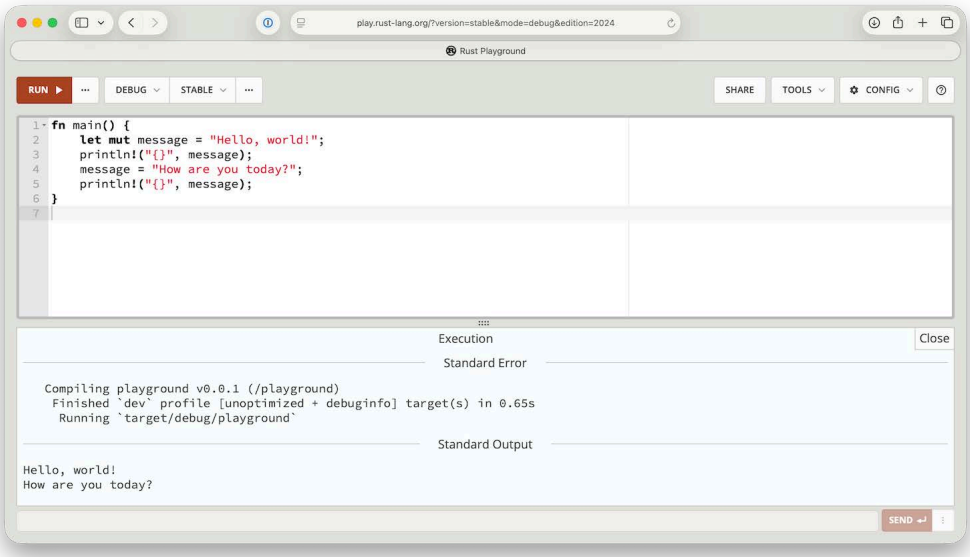
Learn about

- How to run your first Rust program in the Playground
- How to use the `println!` macro
- How to print the value of a variable
- The difference between immutable and mutable variables

In order to get your first taste of Rust programming you don't even need to install anything. Instead, point your web browser at the Rust Playground[1]. You will see a editor-like text area with some Rust source code, along with some buttons to perform operations.

[1] https://play.rust-lang.org

Figure 1.1. The Rust Playground



If the text area contains anything, select it all with the mouse or a keyboard shortcut and delete it; we want to start with a clean slate.

Enter the following program into the text area:

Example 1.1. The classic "Hello, world!" program in Rust

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

Then click the Run button above the editor text area and to the left. You should see another area below the editor, containing messages from the Rust compiler, followed by the output of the program. Under the heading Standard Error you will see something lke this:

```
Compiling playground v0.0.1 (/playground)
```

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.66s
     Running `target/debug/playground`
```

Below that, under the heading Standard Output you should see the output:

```
Hello, world!
```

If this is not the case, then you will see some error messages from the Rust compiler listed under Standard Error. Fix them as you best can, and try again.

The program above is the classic "Hello, world!" program, originally devised by Brian Kernighan [Ker2019]. It has become the standard as the very first program you write when you're learning a new programming language. It lets you test that your setup works, so that you can move on to greater things. Let's analyze the Rust version of this (very short) program line by line.

Every Rust program must have an entry point called `main`, defined as a Rust function using the `fn` keyword. This function has no parameters, but the parameter list (in parentheses) must always be there, even if it is empty. The function header is followed by the function body, in the curly brackets `{` and `}`. The body can contain statements and expressions. In this program the only statement is an invocation of the `println!` macro—you can tell that it is a macro from the exclamation mark after its name. Later we will meet many more Rust macros.

When the program is compiled, the `println!` macro will be expanded into more Rust source code. Here it prints the text `Hello, world!`, expressed here as a static string enclosed in quotes, to the standard output, with an added newline. The semicolon marks the end of the statement. There is also another version of this macro without the trailing newline, called `print!`.

Let's try something else. Change the program in the playground to read:

Example 1.2. Printing a variable in Rust

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

The let expression declares a named variable, which here is called `message`. The `println!` macro can also print the value of a variable instead of just a literal string, provided that there is a placeholder in the format string that can be replaced by the value of the variable. When you run this program, it displays the exact same text as the earlier one.

We will be seeing more than enough error messages by the compiler later on, but just to get an idea of how they look, add a line to the program to change the value of the `message` variable and print it again:

Example 1.3. Trying to assign to an immutable variable

```
fn main() {
    let message = "Hello, world!";
    println!("{}", message);
    message = "How are you today?";
    println!("{}", message);
}
```

If you now run the program, you will find that it does not even compile, let alone run. Instead, you will get a lengthy error message:

```
Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `message`
 --> src/main.rs:4:5
  |
2 |     let message = "Hello, world!";
  |         ------- first assignment to `message`
3 |     println!("{}", message);
4 |     message = "How are you today?";
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut message = "Hello, world!";
  |         +++
```

The Rust compiler has found a problem, reported about it and even offered a way to fix it. The problem is that the `let` keyword defines an *immutable* variable. This is the default in Rust. Once an immutable variable is initialized,

it cannot be changed, or "mutated". To be able to change the value later, the variable needs to be defined with an additional `mut` keyword, as suggested by the compiler:

Example 1.4. Changing the variable to be mutable

```
fn main() {
    let mut message = "Hello, world!";
    println!("{}", message);
    message = "How are you today?";
    println!("{}", message);
}
```

Now there are no compiler errors, and the output is as expected:

```
Hello, world!
How are you today?
```

Note that when the `message` variable was defined, there was no indication of its data type. However, Rust is a statically typed language, and the type of every variable must be known and fixed at the time the program is compiled, either by specifying it or letting the compiler infer it from the context.

We were able to leave out the data type because Rust has *type inference*, meaning that it can decide the data type of the variable based on the value used to initialize it. In most cases the inferred data type is exactly what we intended, but in some situations (which we will see later) Rust needs the data type to be explicitly specified. In this case the text in quotes can only be a string.

The Playground is essentially the Rust compiler and the Cargo tool packaged into a web application that can be driven with your web browser. We will have a look at both the compiler and Cargo soon, but first let's make a couple of small Rust programs just to get familiar with some of the basic building blocks.

You can create much more elaborate programs with the Playground than just the traditional "Hello, world!", because it also provides you with some of the most common Rust libraries. It also has many of the utilities that are part of the standard Rust tools, such as `rustfmt` for formatting your Rust source code

according to the guidelines in the Rust style guide, and Clippy for checking your code for common mistakes.

The Rust Playground is excellent for getting familiar with Rust, but it does have its limitations. At some point you will want to start working on Rust programs with tools installed locally on your computer. In the next chapter we will have a look at how to download, install and test drive the official Rust development tools.

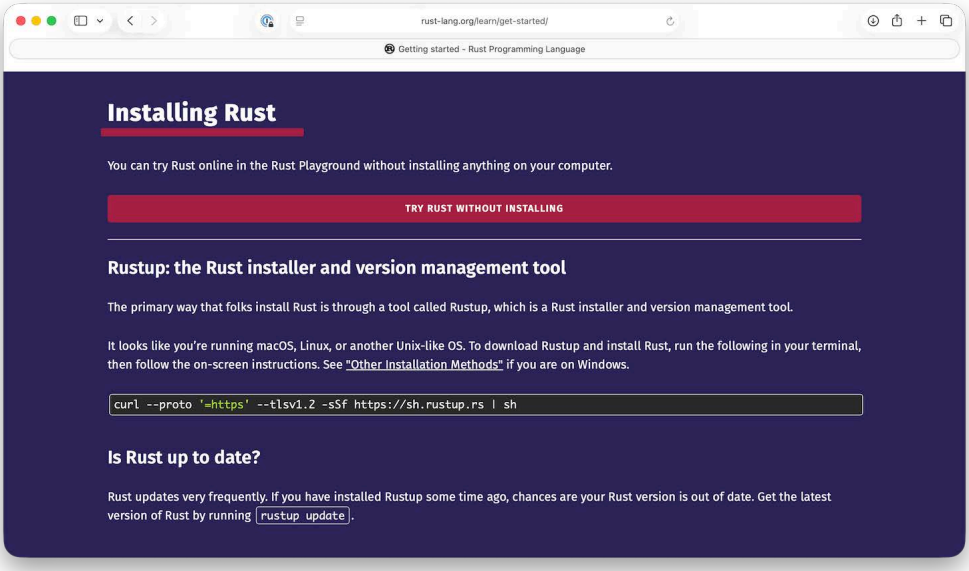
Chapter 2. Getting the Rust tools

Learn about

- How to download and install the Rust development tools
- How to update the tools after installation

When you are ready to download the Rust development tools on your computer and start to use them locally, go to the Rust home page[1] and click the Get Started button. You will be directed to the official method of installing Rust, which is done using a tool called `rustup`. This tool will need to be downloaded first and executed, in a manner that is dependent on the operating system on your computer.

Figure 2.1. Installing Rust



[1] https://rust-lang.org

The Rust home page will most likely detect your operating system, and offer you a suitable way of getting `rustup`. On Unix-like systems, like Linux and macOS, you will use `curl` to download a shell script and execute it. You can also download the shell script using the URL shown, and audit it before executing it. On Windows you will need to download the `rustup-init.exe` program, which requires the Visual C++ Build Tools. There are also standalone installers, so you will need to make a choice that best suits your environment. The Rust documentation has plenty of up-to-date material about installation on various platforms.

For the sake of discussion, let's assume that you are running Linux or macOS, and you execute the `rustup` initialization script. This will result in the `rustup` tool being installed, and then used to download and install the rest of the Rust tools, including the Rust compiler. The `rustup` tool will also be your method of keeping the Rust tools up to date, and it will be installed in a directory that ends up in your search path.

After the installation is complete, you can use the command `rustup show` to check that the Rust toolchain has been installed. On the author's machine (Apple MacBook Air M4), at the time of this writing, the result looks like this:

```
% rustup show
Default host: aarch64-apple-darwin
rustup home:  /Users/me/.rustup

installed toolchains
--------------------
stable-aarch64-apple-darwin (active, default)

active toolchain
----------------
name: stable-aarch64-apple-darwin
active because: it's the default toolchain
installed targets:
  aarch64-apple-darwin
  x86_64-unknown-linux-gnu
```

There are several versions of the Rust toolchain, with the "stable" version being the latest official release version. Since Rust is developing all the time, there are also "nightly" versions that you can use if you want to use features

that may not be ready for prime time yet. In this book we will only use the stable toolchain.

Even the stable toolchain gets updates every six weeks. You can use `rustup` to check for them and install them. At the time of writing this, a new version of the stable toolchain had just been released, so the `rustup update stable` command resulted in downloading and installing the new version 1.92.0:

```
% rustup update stable
info: syncing channel updates for 'stable-aarch64-apple-darwin'
info: latest update on 2025-12-11, rust version 1.92.0 (ded5c06cf 2025-12-08)
info: downloading component 'rust-std' for 'x86_64-unknown-linux-gnu'
 28.0 MiB /  28.0 MiB (100 %)   4.5 MiB/s in  6s
info: downloading component 'rust-src'
  3.4 MiB /   3.4 MiB (100 %)   2.6 MiB/s in  1s
info: downloading component 'cargo'
  8.2 MiB /   8.2 MiB (100 %)   2.8 MiB/s in  3s
info: downloading component 'clippy'
  2.8 MiB /   2.8 MiB (100 %)   1.2 MiB/s in  2s
info: downloading component 'rust-docs'
 20.5 MiB /  20.5 MiB (100 %)   3.6 MiB/s in  7s
info: downloading component 'rust-std'
 26.0 MiB /  26.0 MiB (100 %)   2.7 MiB/s in  9s
info: downloading component 'rustc'
 60.9 MiB /  60.9 MiB (100 %)   3.4 MiB/s in 19s
info: downloading component 'rustfmt'
  1.5 MiB /   1.5 MiB (100 %)   1.3 MiB/s in  1s
info: removing previous version of component 'rust-std' for 'x86_64-unknown-linux-gnu'
info: removing previous version of component 'rust-src'
info: removing previous version of component 'cargo'
info: removing previous version of component 'clippy'
info: removing previous version of component 'rust-docs'
info: removing previous version of component 'rust-std'
info: removing previous version of component 'rustc'
info: removing previous version of component 'rustfmt'
info: installing component 'rust-std' for 'x86_64-unknown-linux-gnu'
 28.0 MiB /  28.0 MiB (100 %)  26.5 MiB/s in  1s
info: installing component 'rust-src'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 20.5 MiB /  20.5 MiB (100 %)   5.0 MiB/s in  2s
```

```
info: installing component 'rust-std'
info: installing component 'rustc'
 60.9 MiB /  60.9 MiB (100 %)  27.8 MiB/s in  2s
info: installing component 'rustfmt'

  stable-aarch64-apple-darwin updated - rustc 1.92.0 (ded5c06cf 2025-12-08) (from rustc
 1.91.1 (ed61e7d7e 2025-11-07))

info: checking for self-update
```

The output shows that all the Rust tools were updated. As a last step, `rustup` also tried to update itself, but there was no update available for it. You may want to issue the `rustup update stable` command from time to time to get the latest tools. Check the Rust language blog on the home page regularly for news about new releases, features and bug fixes.