

The title is composed of four stacked, overlapping rectangular blocks. The top block is yellow and contains the word 'Learn'. The second block is orange and contains the word 'Rust'. The third block is red and contains the word 'Programming'. The bottom block is blue and contains the word 'Today'. The blocks are offset to the left, creating a stepped effect. The background is a light green gradient.

Learn

Rust

Programming

Today

JERE KÄPYAHO

Learn Rust Programming Today

Learn Rust Programming Today

From Building Blocks to a Working Application

Jere Käpyaho

Työn automaattinen analysointi tietojen saamiseksi erityisesti kuvioista, suuntauksista ja korrelaatioista ("tekstin ja tietojen hakeminen") on kielletty.

Copyright © 2026 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

Introduction	xvii
Rust in a nutshell	xviii
Notational differences	xviii
Conceptual differences	xix
Learning Rust by doing	xix
Who this book is for	xx
Source code repositories	xxi
Acknowledgments	xxi
I Getting Started with Rust	1
1 Using the Rust Playground	5
2 Getting the Rust tools	11
3 Using the Rust compiler	15
4 Editors for Rust source code	17
Editing Rust source code	17
Visual Studio Code	17
Vim, Neovim, and Emacs	18
Zed	19
RustRover	19
5 Getting to know Cargo	21
Hello, Cargo	21
The Cargo configuration file	23
II Building Blocks of Rust Programs	25
6 Strings and numbers	31
Storing text and numbers	31
Strings	31
String slices	33
Numbers	34
7 Tuples and arrays	39
Storing data in tuples and arrays	39
Tuples	39
Arrays	40
Sorting the array	44
8 Iteration and looping	47
Looping through the elements	47
9 Making decisions	51

Truth values	51
Deciding which statements to execute	51
10 Functions	57
Defining and using functions	57
11 Structs	63
Grouping values into structs	63
Initializing structs	65
Struct implementations	66
Methods	69
12 Enums and pattern matching	73
Using enums for alternative values	73
Enums with associated values	78
Pattern matching	78
13 Optional values	83
Option instead of nullable	83
Handling optional values	87
14 Ownership, references and borrowing	89
Memory safety	89
Ownership	89
Scope	91
Examples of ownership and scope	91
References	96
Borrowing	97
15 Vectors	99
Storing elements in a dynamic vector	99
Sorting a vector	101
16 Command-line arguments	103
Getting at the arguments	103
Converting a string to a MonthDay	103
Handling the month-day argument	105
17 Errors	109
No panic, but a result	109
Using the unwrap and expect methods	110
Propagating errors	111
18 Traits	115
Traits specify behavior	115
Implementing the Display trait	116
The Default trait	118

The FromStr trait	120
Defining traits	123
19 Crates	125
Beyond the standard library	125
Making random dates with some crates	126
Making random descriptions	130
Making random events	132
20 Testing	135
Testing with Cargo	135
A better Date struct	135
Creating a test module	137
Random dates again	139
Writing tests for the Category struct	140
21 Comments and documentation	143
Documentation is your friend	143
Comments	143
Documentation comments	145
Generating documentation	145
22 The Birthday program	147
Starting the Birthday program	147
Getting the BIRTHDATE environment variable	148
Parsing BIRTHDATE as a date	149
Constructing the age message	152
Adding unit tests	153
III Programming Today	157
23 Creating the Today program	163
Starting the Today program with Cargo	163
The revised Event struct	164
The revised Category and MonthDay structs	165
Testing the Today program	167
24 Modules	169
Packages and crates	169
Modules	169
The birthday module	171
The events module	172
25 Event providers	179
Introducing event providers	179
Creating a new module for the providers	179

A simple event provider for testing	180
Creating submodules for providers	183
26 Text file event provider	187
Putting events in a text file	187
Developing a text file event provider	188
Integrating the text file provider	193
27 CSV file event provider	195
The CSV file format	195
Using the csv crate	196
Integrating the CSV file provider	199
28 Configuration directories	201
Everything in its right place	201
Getting the configuration directory name	202
Building the configuration path	203
Placing the data files	205
29 Configuration file	207
Using a TOML file to store the configuration	207
Parsing the TOML file	208
Instantiating the event providers	211
Preventing duplicate event providers	215
30 Reading events from a SQLite database	217
Adding a database to the program	217
Getting SQLite	218
The event database schema in SQLite	218
Using the rusqlite crate	221
Testing database operations	224
Getting events from the database	226
Integrating the SQLite event provider	228
31 Reading events from a web service	231
Reading data from the web	231
The web service and REST API	231
Testing the API with curl	231
Making the web event provider	233
Making an HTTP request with the request crate	234
Preparing for the web event provider	237
32 Filtering events at the source	239
Getting just the events we want	239
Introducing event filters	239

Filter options	239
Accepting the event	240
Constructing filters with the Builder pattern	242
Testing the filters and the builder	244
Refactoring the event providers	246
Refactoring the run function	251
33 Handling command-line arguments	255
Controlling the program from the command line	255
Parsing command-line arguments with clap	257
Defining subcommands	261
34 Adding new event kinds	265
More than just singular events	265
Annual events	265
Updating event providers to support annual events	267
Rule-based events	270
Resolving the rule	277
Updating the CSV file event provider to support rule-based events	284
35 Logging	287
Leaving a log of the execution	287
Adding the logging crates	287
Logging levels and macros	288
Converting println! output to logging macros	290
36 Adding new events	293
New events from the command line	293
Revising the EventProvider trait	294
Implementing event addition for TextFileProvider	294
Implementing event addition for CSVFileProvider	296
Command-line arguments for adding events	298
Adding events to the SQLite database	302
Trying to add an event from the command line	304
Updating the providers subcommand	305
Adding an event (finally)	307
37 Grouping and sorting the events	309
All the events, grouped and sorted	309
Partitioning the events	312
38 Today 1.0	315
First public version of Today	315

Building a release version	315
Installing the program	315
Configuring the program	316
IV Going Further than Today	319
39 Refactoring the Event struct	323
A different kind of Event	323
Focusing on the date	323
Simplified	328
40 Making an XML event provider	329
Making an XML event provider	329
Defining an XML Schema	329
Making the XML provider	333
Parsing XML	334
41 One final refactoring	341
Configuration of event provider activity	341
Introducing the event manager	342
Checking for active providers	344
Updating the provider information	348
Final main function	351
42 Future development ideas	355
Localized category names	355
Category exclusions	355
Year count for events	355
Showing upcoming events	355
Color terminal display	356
A About this book	357
B About the author	359
Further reading	361
Index	363

List of Examples

1.1 The classic "Hello, world!" program in Rust	6
1.2 Printing a variable in Rust	7
1.3 Trying to assign to an immutable variable	8
1.4 Changing the variable to be mutable	9
3.1 The traditional "Hello, world!" program	15
5.1 The Cargo.toml configuration file	24
6.1 Using a string variable	32
6.2 Using the println! macro	32
6.3 Using a string slice	33
6.4 Using an integer variable	34
6.5 Using floating-point variables	36
7.1 Using a tuple variable	39
7.2 Swapping the components of two tuples	40
7.3 Making an array with tuple values as elements	41
7.4 Sorting an array of tuples	44
8.1 Iterating the elements of an array with a while loop	47
8.2 Using an iterator and the for statement on the array	49
9.1 Listing events that match a target date	53
10.1 A function to extract year, month, and day	57
10.2 Using the get_year_month_day function	59
11.1 The Date struct	64
11.2 The Event struct	64
11.3 Replacing tuples with structs	64
11.4 Utility functions to make dates and events	67
11.5 Using the field init shorthand	67
11.6 An impl block for the Date struct	68
11.7 An impl block for the Event struct	69
11.8 A helper struct and a related accessor function	69
11.9 The complete events program	70
12.1 An enum representing the months	74
12.2 Date struct using the Month enum	74
12.3 The events program updated to use the Month enum	76
12.4 Day count without leap year checking	79
12.5 Day count with less match arms	79
12.6 Day count with leap year check	80

12.7 Implementing a method for an enum	81
13.1 Category struct with optional field	84
13.2 The impl block for the Category struct	84
13.3 Event struct with a category field	85
13.4 Event declarations updated to include category	85
13.5 Handling the optional second category component	87
14.1 Trying to use a variable that is out of scope	92
14.2 Using a variable in the current scope	93
14.3 Trying to assign a type that does not implement the Copy trait	93
14.4 Deriving the Copy trait	94
14.5 Deriving both Copy and Clone along with Debug	96
14.6 Multiple immutable references	96
14.7 Mutable references to a variable	97
15.1 Declaring a vector and adding elements	99
15.2 Using the vec! macro to initialize a vector	100
16.1 Making a MonthDay value from a string	103
16.2 Making a MonthDay value from an integer	104
16.3 Handling the month-day argument	105
17.1 Trying to open a file	110
17.2 Using the unwrap method to handle the error	111
17.3 Using the expect method to handle the error	111
17.4 Propagating errors from a function	112
18.1 Using the derive attribute	115
18.2 Implementing the Display trait for Category	116
18.3 Printing out struct values normally	117
18.4 Display trait for Event	117
18.5 Implementing the Default trait	118
18.6 Testing the Display and Default traits	119
18.7 Implementing the FromStr trait	121
18.8 Implementing FromStr for MonthDay	122
18.9 Dealing with Result when parsing MonthDay	123
19.1 Method for checking the validity of a date	126
19.2 Checking some good and bad dates	127
19.3 Generating random dates	129
19.4 Testing random date generation	129
19.5 Generating random sentences	131
19.6 Generating a random event	132
19.7 The first ten of 10,000 random events	133

19.8 A sampling of randomly generated events	133
20.1 Improving the Date struct	136
20.2 New random function for Date	137
20.3 Unit tests for the Date struct	138
20.4 Making random dates again	139
20.5 Unit tests for the Category struct	140
21.1 Example of a line comment	144
21.2 Example of a block comment	144
21.3 Example of a documentation comment	145
22.1 Generating the Birthday program	147
22.2 Getting the BIRTHDATE value	148
22.3 Handling the result of date parsing	150
22.4 Stub implementation of the make_message function	151
22.5 The complete make_message function	152
22.6 Testing the birthday message generation	153
23.1 Initial Cargo.toml file for Today	163
23.2 The EventKind enum and the revised Event struct	164
23.3 Singular event constructor and helpers	164
23.4 Improved FromStr implementation for Category	165
23.5 Improved FromStr implementation for MonthDay	166
23.6 Testing the initial program version	167
24.1 Initial contents of lib.rs	170
24.2 The revised main function	170
24.3 The birthday module	171
24.4 Invoking the handle_birthday function	171
24.5 The new birthday module	172
24.6 Invoking handle_birthday from the birthday module	172
24.7 Accessor functions for Event	173
24.8 Complete listing of src/lib.rs up to this point	174
24.9 Complete listing of src/events.rs up to this point	175
25.1 The EventProvider trait in src/providers/mod.rs	180
25.2 A simple event provider	181
25.3 Using the event providers in lib.rs	182
25.4 Example of a nested module in src/providers.rs	183
25.5 An event provider in a separate module	184
26.1 Events in a plain text file	187
26.2 Text file event provider	188
26.3 EventProvider trait implementation for text files	189

26.4 Possible states of reading the event text file	190
26.5 The <code>get_events</code> method implementation for <code>TextFileProvider</code>	190
26.6 Making use of the text file event provider	193
27.1 List of events as a CSV file	195
27.2 CSV file provider	196
27.3 <code>EventProvider</code> trait implementation for the CSV file provider	197
27.4 Using the CSV file provider	199
28.1 Helper program to print out the configuration directory	202
28.2 Helper function to get the configuration directory	204
28.3 Constructing the configuration directory	204
28.4 Constructing paths to data files	205
29.1 The configuration file in TOML format	207
29.2 Project file with <code>toml</code> and <code>serde crates</code> added	208
29.3 Configuration data structures	209
29.4 Updated <code>run</code> function signature	209
29.5 Parsing the TOML configuration file in <code>main.rs</code>	210
29.6 Instantiating the event providers in <code>src/lib.rs</code>	213
29.7 Revised <code>run</code> function implementation	214
29.8 Checking for existing event providers by name	215
30.1 Database schema SQL statements	219
30.2 Adding a category and some events to the database	220
30.3 SQLite event provider implementation	221
30.4 Type aliases for category ID and hash map	222
30.5 Getting the categories from the database	223
30.6 Testing the database operations	225
30.7 Implementing the <code>EventProvider</code> trait for <code>SQLiteProvider</code>	226
30.8 Getting events from the SQLite database	227
30.9 Testing the <code>get_events</code> method of <code>SQLiteProvider</code>	228
30.10 Adding an SQLite event provider to the configuration	228
31.1 The <code>Cargo.toml</code> entries for web-related crates	234
31.2 Web event provider implementation	234
31.3 <code>EventProvider</code> implementation for <code>WebProvider</code>	235
31.4 A helper for JSON parsing	236
31.5 The rest of the <code>get_events</code> method	236
32.1 The <code>EventFilter</code> struct with accessors	240
32.2 The <code>accepts</code> method of the event filter	241
32.3 The filter builder	243
32.4 Using the filter builder	244

32.5 Testing the event filter	245
32.6 Testing the filter builder	245
32.7 The refactored EventFilter trait	247
32.8 Constructing the WHERE clause	248
32.9 Adding the WHERE clause to the helper function	249
32.10 Testing the WHERE clause generation	249
32.11 Web provider refactored for filtering	250
32.12 The new and improved run function	252
33.1 Printing out the command-line arguments	255
33.2 Using clap to process command-line arguments	258
33.3 Making the date argument optional	259
33.4 Constructing an event filter from the command-line arguments ...	260
33.5 Defining subcommands with the clap crate	261
33.6 Using match to handle subcommands	262
33.7 Helper types for provider configuration	263
34.1 Adding a new event kind variant	265
34.2 New constructor for creating annual events	266
34.3 Display trait implementation for Event	267
34.4 Event text file with yearless dates	267
34.5 Configuration entry for yearless event provider in today.toml	268
34.6 Creating a singular or annual event in get_events	269
34.7 Adding a new event kind for rule-based events	270
34.8 Constructing a rule-based event	271
34.9 Defining a struct for the event rule	271
34.10 Enumerated type for the ordinals	272
34.11 Enumerated type for the weekdays	272
34.12 The parse function in the Rule struct	273
34.13 Enriching the Ordinal and Weekday enums with the strum crate	275
34.14 Parsing the event rule	276
34.15 Parsing the rest of the rule	277
34.16 Handling all the EventKind enum variants	278
34.17 The year and month_day methods of Rule	278
34.18 Tests for the event rule	279
34.19 Stub method for resolving the rule	280
34.20 Resolving the rule to a date	281
34.21 Computing the n th weekday of the month	281
34.22 Computing the last weekday of the month	282
34.23 Converting Weekday to chrono::Weekday	282

34.24 More unit tests for rule resolution	283
34.25 Checking for a rule-based event	284
35.1 Preparing for logging in the main function	288
35.2 Adding some debug output to the program	289
35.3 Finding out and printing the configuration directory	290
35.4 Properly logging the configuration file search	291
36.1 The revised EventProvider trait	294
36.2 Adding an event to a text file	295
36.3 Adding an event to a CSV file	296
36.4 Command-line arguments for adding an event	298
36.5 Handling the add subcommand	299
36.6 The add_event function in lib.rs	299
36.7 Function to create the event providers	300
36.8 The add_event function	301
36.9 The find_category_id function	302
36.10 The EventProvider implementation for SQLiteProvider	303
36.11 The insert_event function	304
36.12 Showing the listing of event providers	306
36.13 Adding the kind for event providers	306
37.1 The revised run function	309
37.2 Pluralizing the group headings	310
37.3 Using the partition method of an iterator	312
38.1 A minimal today.toml configuration file	316
38.2 Rust tool releases in a CSV file called rustopedia.csv	317
39.1 Refactoring Event by the date	323
39.2 Constructing an Event with an EventDate	324
39.3 Refactoring the year and month_day methods	324
39.4 Adding the is_singular method	325
39.5 Creating a new-style event in the text file provider	326
39.6 Adding a new-style event to a text file	327
39.7 Implementing the Display trait for EventDate	328
39.8 Collecting events with partition and closure	328
40.1 The XML Schema for the event document	330
40.2 An instance document with some XML-related events	331
40.3 Placeholder implementation of XmlFileProvider	333
40.4 Parsing XML events	335
40.5 Parsing the many forms of event dates in EventDate::parse	338
41.1 The is_active setting for an event provider	341

41.2 The new configuration setting	342
41.3 EventManager partial implementation	342
41.4 New is_active method for the EventProvider trait	345
41.5 New struct field to hold provider active status	345
41.6 Checking if the event provider is active	346
41.7 Checking for active and add support in a provider	347
41.8 Support for getting provider information	348
41.9 Implementing the providers subcommand	349
41.10 Implementing the add subcommand	350
41.11 The revised run function	350
41.12 The revised main function	352

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 Index¹ (June 2026, 12th).
- Rust is now core part of the Linux kernel² (not without controversy, of course).
- Microsoft is reducing the attack surface of shipping products using Rust or lessons learned from it³.
- Amazon Web Services has developed the Bottlerocket container operating system mostly in Rust⁴.
- AdaCore is diversifying from Ada and SPARK to include Rust⁵.

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 repository⁶ 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 repository⁷. 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

1 Using the Rust Playground	5
2 Getting the Rust tools	11
3 Using the Rust compiler	15
4 Editors for Rust source code	17
Editing Rust source code	17
Visual Studio Code	17
Vim, Neovim, and Emacs	18
Zed	19
RustRover	19
5 Getting to know Cargo	21
Hello, Cargo	21
The Cargo configuration file	23

Chapter 1. Using the Rust Playground

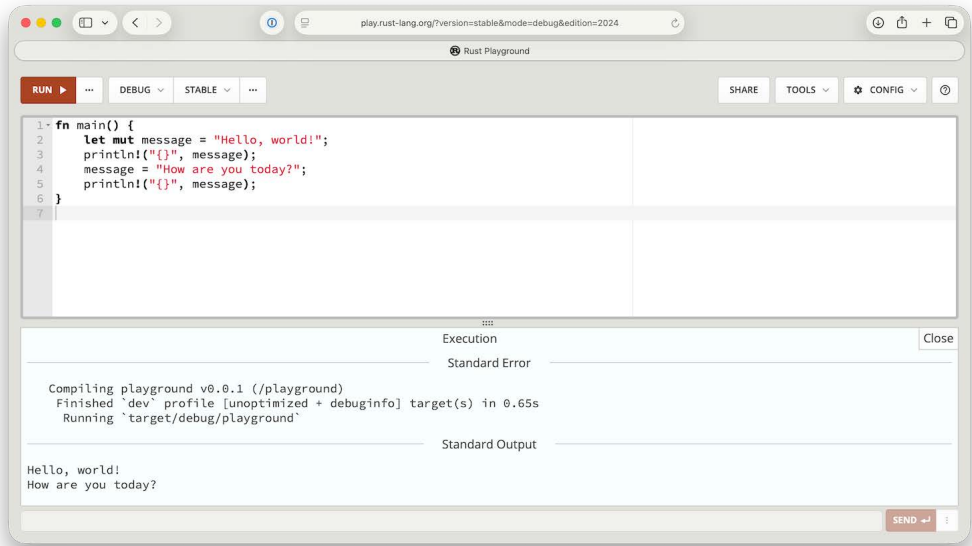
Learn about

- 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 Playground¹. 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 like 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.

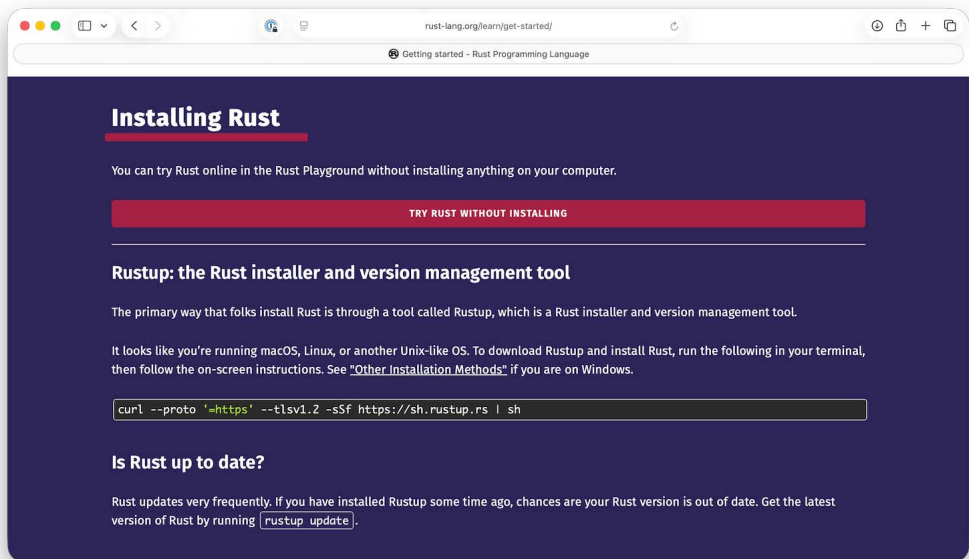
Chapter 2. Getting the Rust tools

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 page¹ 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
```

Chapter 2. Getting the Rust tools

```
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.