🦀 **Rust Guide Installation/Running**

Dylan Green, Noah Holman, Wonda White  
21th September, 2024

# Table of Contents

[Table of Contents 2](#_4lqp25cx7kth)

[Installing 3](#_ovawxjoi07ia)

[Running 6](#_lzr2si3oshbv)

[Sources 8](#_ab87cgpufwn7)

[Example Code 9](#_cytftoi1c1rj)

[Hello World 9](#_l22e31kxo364)

[Looping 9](#_gz0w4h88ia23)

[Data Types 10](#_fibyn1pstwg9)

[Read Input 11](#_32hu5x9krbgh)

[Math 12](#_waf7xboq1ttp)

[If 13](#_ific5dpkz20v)

[Functions 14](#_vglzzqu1pinw)

Transformations [15](#_vglzzqu1pinw)

# 

# Installing

The recommended way to install Rust is to download it through *rustup*, a command line tool for managing Rust versions and associated tools. You’ll need an internet connection to download it. In this guide we will mainly cover Linux, MacOS, and Windows but there are options for other systems.

**For Linux/MacOS Enter the following into the terminal.**

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

You will need a linker, if you don’t have one, install a C compiler. 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.

**For MacOS users you can run the following for a C compiler.**

$ xcode-select --install

Then you just follow the instructions on the console to install with the settings you want for rustup.

**For Windows users, go to the following link.**

<https://www.rust-lang.org/tools/install>

You download the executable provided and select your install options in the console window. At some point in the installation, you’ll be prompted to install Visual Studio. This provides a linker and the native libraries needed to compile programs. These are necessary for Rust to run in the windows command line or VsCode/Codium so Visual Studio has to be installed.

**If you prefer not to use shell script you can directly download rustup-init here.**

[https://forge.rust-lang.org/infra/other-installation-methods.html](https://forge.rust-lang.org/infra/other-installation-methods.html" \l "other-ways-to-install-rustup)

There you can also find the official Rust standalone installers that contain a single release of Rust, and are suitable for offline installation. They come in three forms: tarballs (extension .tar.xz), that work in any Unix-like environment, Windows installers (.msi), and Mac installers (.pkg). These installers come with rustc, cargo, rustdoc, the standard library, and the standard documentation, but do not provide access to additional cross-targets like rustup does.

If you want to build the Rust toolchain from source code, Rust language also provides downloads for source code tarballs.

**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 Terminal, enter the following.

> echo %PATH%

In PowerShell, enter the following.

> echo $env:Path

In Linux and macOS, enter the following into the console.

$ echo $PATH



**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!



# Running

Like before this will cover Linux, MacOS, and Windows.

To start it is recommended that you create a project directory to store your Rust code. It doesn’t matter to Rust where your code lives. 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

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 this simple “Hello World” code.**

|  |
| --- |
| **fn** **main**() {  println!("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.

**Windows users have an option besides the command line to run Rust.**

You can open VsCode/Codium and download the extension Rust Analyzer from the extensions search bar, links below to downloads if needed.

<https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer>

<https://rust-analyzer.github.io/>

**Open VsCode/Codium and open a folder you want to make your Rust program in. Then run the following example code in a new terminal to create a new project with cargo.**

cargo new hello\_world

This will create a simple Hello World main.rs program along with a default Cargo.toml dependency file in the new directory.

**You can use this by navigating to the new directory, and building/running with the following commands.**

cd hello\_world

cargo build

cargo run

# Sources

* [The Rust Programming Language - https://doc.rust-lang.org/book/title-page.html](https://doc.rust-lang.org/book/title-page.html)
* [Rust Forge - https://forge.rust-lang.org/index.html](https://forge.rust-lang.org/index.html)
* [Visual Studio Code - https://code.visualstudio.com/docs/languages/rust](https://code.visualstudio.com/docs/languages/rust)

# Example Code

## Hello World

|  |
| --- |
| **fn** **main**(){  println!("Hello, World!"); } |

## Looping

|  |
| --- |
| **fn** **main**() {  // Infinite loop with break condition  println!("Loop:");  **let** **mut** count = 0;  **loop** {  **if** count == 5 {  **break**; // Exit the loop when count is 5  }  println!("count = {}", count);  count += 1;  }   // While loop  println!("\nWhile loop:");  **let** **mut** number = 0;  **while** number < 5 {  println!("number = {}", number);  number += 1;  }   // For loop  println!("\nFor loop:");  **for** i **in** 0..5 { // i ranges from 0 to 4  println!("i = {}", i);  }   // Iterating over a collection using for loop  println!("\nFor loop with collection:");  **let** fruits = vec!["apple", "banana", "cherry"];  **for** fruit **in** &fruits {  println!("fruit = {}", fruit);  } } |

## Data Types

|  |
| --- |
| **fn** **main**() {  // Integer (i32 by default)  **let** int\_num: i32 = 42;  println!("Integer: {}", int\_num);   // Floating-point number (f64 by default)  **let** float\_num: f64 = 3.1415;  println!("Floating-point: {}", float\_num);   // Boolean  **let** is\_true: bool = true;  println!("Boolean: {}", is\_true);   // Character  **let** char\_val: char = 'A';  println!("Character: {}", char\_val);   // String (String is a heap-allocated type)  **let** string\_val: String = String::from("Hello, Rust!");  println!("String: {}", string\_val);   // Array (fixed size, all elements must be of the same type)  **let** array: [i32; 3] = [1, 2, 3];  println!("Array: {:?}", array);   // Tuple (can contain values of different types)  **let** tuple: (i32, f64, char) = (10, 4.5, 'Z');  println!("Tuple: {:?}", tuple);   // Slice (view into an array, dynamically sized)  **let** array\_slice: &[i32] = &array[1..];  println!("Slice of array: {:?}", array\_slice);   // Option type (used for values that may or may not exist)  **let** some\_value: Option<i32> = Some(5);  **let** no\_value: Option<i32> = None;   // Result type (used for error handling)  **let** ok\_value: Result<i32, &str> = Ok(10);  **let** error\_value: Result<i32, &str> = Err("An error occurred"); } |

## Read Input

|  |
| --- |
| **use** std::io;  **fn** **main**() {  // Create a mutable string to hold the user input for the name  **let** **mut** name = String::new();  println!("Enter your name:");    // Read the user's input for the name  io::stdin()  .read\_line(&**mut** name)  .expect("Failed to read input");    // Trim the newline character from the input  **let** name = name.trim();   // Create a mutable string to hold the user input for the age  **let** **mut** age = String::new();  println!("Enter your age:");   // Read the user's input for the age  io::stdin()  .read\_line(&**mut** age)  .expect("Failed to read input");    // Convert the input from string to integer  **let** age: u32 = age.trim().parse().expect("Please type a valid number");   // Display the output with the input values  println!("Hello, {}! You are {} years old.", name, age); } |

## Math

|  |
| --- |
| fn main() {  // addition  let sum = 5 + 10;  println!("The value of sum is: {sum}");   // subtraction  let difference = 95.5 - 4.3;  println!("The value of difference is: {difference}");   // multiplication  let product = 4 \* 30;  println!("The value of product is: {product}");   // division  let quotient = 56.7 / 32.2;  println!("The value of quotient is: {quotient}");  let truncated = -5 / 3; // Results in -1  println!("The value of truncated is: {truncated}");   // remainder  let remainder = 43 % 5;  println!("The value of remainder is: {remainder}"); } |

## If

|  |
| --- |
| fn main() {  let number = 6;   //Basic if true else false  if number < 5 {  println!("condition was true");  } else {  println!("condition was false");  }     //Not equals statement  if number != 0 {  println!("number was something other than zero");  }    //Divisibility if statement  if number % 4 == 0 {  println!("number is divisible by 4");  } else if number % 3 == 0 {  println!("number is divisible by 3");  } else if number % 2 == 0 {  println!("number is divisible by 2");  } else {  println!("number is not divisible by 4, 3, or 2");  }     let condition = true;   //If Let statement  //If condition True, num = 5, else num = 6  let number = if condition { 5 } else { 6 };  println!("The value of number is: {number}");  } |

## Functions

|  |
| --- |
| fn main() {  //Function Call  another\_function();   //Passing values to function  print\_labeled\_measurement(5, 'h');   //Return value  let x = five();  println!("The value of x is: {x}");  }  //Basic print function fn another\_function() {  println!("Another function."); }  //Function with parameter fn print\_labeled\_measurement(value: i32, unit\_label: char) {  println!("The measurement is: {value}{unit\_label}"); }  //Declare return value fn five() -> i32 {  5 } |

## Transformations

|  |
| --- |
| /\* //This code does not compile //This example shows how the compilier helps to find errors  fn main() {  let x = 5;  println!("The value of x is: {x}");  x = 6;  println!("The value of x is: {x}"); }  \*/   **fn** **main**() {  //This code has x as a mutable variable  **let** **mut** x = 5;  println!("The value of x is: {x}");  //Value has been Changed  x = 6;  println!("The value of x is: {x}");   //Constants are immutable always and last for the runtime  **const** THREE\_HOURS\_IN\_SECONDS: u32 = 60 \* 60 \* 3;  println!("The value of 3 hours in seconds is:   {THREE\_HOURS\_IN\_SECONDS}");    /\*Shadowing performs a transformation but the values are  immutable afterwords \*/  **let** y = 5;   **let** y = y + 1;   {  **let** y = y \* 2;  println!("The value of y in the inner scope is: {y}");  }   println!("The value of y is: {y}");  } |