|  |
| --- |
| Learning Rust |
|  |
| August, 2025The Rust Programming Language (AKA The Rust book) – [Interactive version](https://rust-book.cs.brown.edu/experiment-intro.html) | |

* The non-experimental version of the book is available offline with installations of Rust made with rustup; run rustup doc --book to open.
* Foreword:
  + It wasn’t always so clear, but the Rust programming language is fundamentally about *empowerment.* Rust empowers you to reach farther, **to program with confidence in a wider variety of domains than you did before.**
  + See foreword for how.
* Introduction:
  + Helps you write faster, more reliable software.
  + Rust gives you the option to control low-level details (such as memory usage) without all the hassle traditionally associated with such control.
  + Safety *and* productivity, speed *and* ergonomics.
  + Cargo -> the included dependency manager and build tool.
* Getting started:
  + rustup, a command line tool for managing Rust versions and associated tools.
  + Also need a linker to join compiled outputs into one file. Typically included in C compilers. A C compiler is also useful because some common Rust packages depend on C code and will need a C compiler.
  + rustc -> Rust compiler
  + Check version: rustc –version
  + rustc <filename>.rs to compile a Rust program
  + The main function in a program is special: it is always the first code that runs in every executable Rust program.
* fn *main*() {
* *println!*("Hello, world!");
* }
  + println! calls a Rust macro. If it had called a function instead, it would be entered as println (without the !). Rust macros are a way to write code that generates code to extend Rust syntax. 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.
  + 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*.)
  + 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. Having multiple files will also be easier. Thus, the vast majority of Rust projects use cargo, and hence we shall too!
    - $ cargo new hello\_cargo
    - $ cd hello\_cargo
  + Cargo.toml file (TOML = *Tom’s Obvious, Minimal Language).* Contains config info to compile our program + dependencies, which are source code files referred to as crates.
  + Cargo enforced a specific folder structure too. Cargo helps you organize your projects. There’s a place for everything, and everything is in its place.
  + cargo build
    - Places executable in ./*target/debug/ since the default build is a debug build.*
  + To compile and run in one command: cargo run
  + cargo check
    - Quickly checks your code to make sure it compiles but doesn’t produce an executable.
    - 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.
  + cargo build --release to compile it with optimizations.
    - But it lengthens the time it takes for your program to compile.
* Programming a Guessing Game:
  + By default, Rust has a set of items defined in the standard library, std, that it brings into the scope of every program. This set is called the *prelude.*
  + To obtain user input and then print the result as output, we need to bring the io input/output library into scope, but this isn’t included in the prelude. Therefore, use std::io;
  + let to declare variables. In Rust, variables are immutable by default, meaning once we give the variable a value, the value won’t change. For mutability, add mut before the variable name.
  + Start a comment with //
  + let mut guess = String::new();
    - The equal sign (=) tells Rust we want to *bind* something to the variable now.
    - String::new() is a function that returns a new instance of a String.
    - [String](https://doc.rust-lang.org/std/string/struct.String.html) is a string type provided by the standard library that is a growable, UTF-8 encoded bit of text.
    - The :: syntax in the ::new line indicates that new is an associated function of the String type. An *associated function* is a function that’s implemented on a type, in this case String.
  + io::stdin().read\_line(&mut guess).expect("Failed to read line");
    - The stdin() function returns an instance of [std::io::Stdin](https://doc.rust-lang.org/std/io/struct.Stdin.html), which is a type that represents a handle to the standard input for your terminal.
    - The & indicates that this argument is a *reference.* For now, all you need to know is that, like variables, references are immutable by default. Hence, you need to write &mut guess rather than &guess to make it mutable.
    - read\_line appends whatever the user enters into the string we pass to it, but it also returns a Result value. [Result](https://doc.rust-lang.org/std/result/enum.Result.html) is an [*enumeration*](https://rust-book.cs.brown.edu/ch06-00-enums.html), often called an *enum*, which is a type that can be in one of multiple possible states. We call each possible state a *variant*.
      * Result’s variants are Ok and Err.
      * An instance of Result has an [expect method](https://doc.rust-lang.org/std/result/enum.Result.html#method.expect) that you can call.
      * If this instance of Result is an Err value, expect will cause the program to crash and display the message that you passed as an argument to expect. If this instance of Result is an Ok value, expect will take the return value that Ok is holding and return just that value to you so you can use it. In this case, that value is the number of bytes in the user’s input.
      * If you don’t call expect, the program will compile, but you’ll get a warning.
      * The right way to suppress the warning is to actually write error-handling code, but in our case we just want to crash this program when a problem occurs, so we can use expect.
  + To print variables vs. expressions: println!("x = {x} and y + 2 = {}", y + 2);
  + [dependencies] rand = "0.8.5"
    - 0.8.5 is a Semantic version specifier. Cargo understands [Semantic Versioning](http://semver.org) (sometimes called *SemVer*), which is a standard for writing version numbers. The specifier 0.8.5 is actually shorthand for ^0.8.5, which means any version that is at least 0.8.5 but below 0.9.0. Cargo considers these versions to have public APIs compatible with version 0.8.5, and this specification ensures you’ll get the latest patch release that will still compile with the code in this chapter
    - When we include an external dependency, Cargo fetches the latest versions of everything that dependency needs from the *registry*, which is a copy of data from [Crates.io](https://crates.io/). Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.
  + Also, say that next week version 0.8.6 of the rand crate comes out and that version contains an important bug fix, but it also contains a regression that will break your code. If someone else then tries to build our code from scratch, they won’t be able to. To handle this, Rust creates/updates the *Cargo.lock* file the first time you run cargo build for each crate. When you build your project in the future, Cargo will see that the *Cargo.lock* file exists and will use the versions specified there rather than doing all the work of figuring out versions again. This lets you have a reproducible build automatically. In other words, your project will remain at 0.8.5 until you explicitly upgrade, thanks to the *Cargo.lock* file.
  + When you *do* want to update a crate, Cargo provides the command update, which will ignore the *Cargo.lock* file and figure out all the latest versions that fit your specifications in *Cargo.toml*. Cargo will then write those versions to the *Cargo.lock* file. In this case, Cargo will only look for versions greater than 0.8.5 and less than 0.9.0.
  + Another neat feature of Cargo is that running the cargo doc --open command will build documentation provided by all your dependencies locally and open it in your browser.
* use std::cmp::*Ordering*;
* use std::io;
* use rand::*Rng*;
* fn *main*() {
* *// --snip--*
* *println!*("You guessed: {guess}");
* let guess: u32 = guess.*trim*().*parse*().*expect*("Please type a number!");
* match guess.*cmp*(&secret\_number) {
* *Ordering*::*Less* => *println!*("Too small!"),
* *Ordering*::*Greater* => *println!*("Too big!"),
* *Ordering*::*Equal* => *println!*("You win!"),
* };
* }
  + The Ordering type is another enum and has the variants Less, Greater, and Equal.
  + A match expression is made up of *arms*. An arm consists of a *pattern* to match against, and the code that should be run if the value given to match fits that arm’s pattern.
  + We create a variable named guess. But wait, doesn’t the program already have a variable named guess? It does, but helpfully Rust allows us to shadow the previous value of guess with a new one. This is called “Shadowing”.
  + The parse method on strings converts a string to another type. Here, we use it to convert from a string to a number. We need to tell Rust the exact number type we want by using let guess: u32.
  + The loop keyword creates an infinite loop.