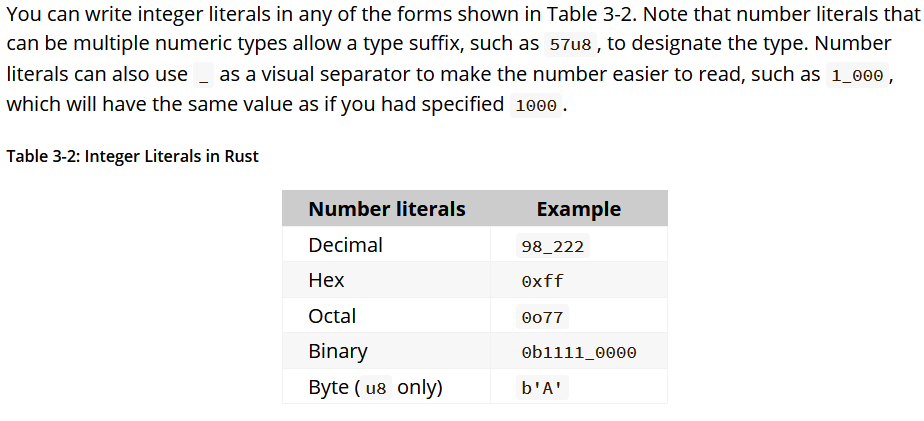
|  |
| --- |
| Learning Rust |
|  |
| August, 2025The Rust Programming Language (AKA The Rust book) – [Interactive version](https://rust-book.cs.brown.edu/experiment-intro.html) | |

* 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, (execution) 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 (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. (If I understand correctly, ‘guess’ is a growable string, and strings grow by copying the new contents into newly-allocated memory, and hence read\_line may change where it points, which we explicitly allow by specifying ‘mut’.)
    - 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 it. 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.
* Common Programming Concepts:
  + By default, variables are immutable. But mutability can be very useful, and can make code more convenient to write. Therefore, you can make them mutable by adding mut in front of the variable name.
  + Like immutable variables, *constants* are values that are bound to a name and are not allowed to change, but there are a few differences between constants and variables.
    - The type of the value they hold *must* be annotated (We’ll cover types and type annotations in the next section).
    - Difference between constants and immutable variables?
      * Constants can be declared in any scope, including the global scope.
      * The last difference is that constants may be set only to a constant expression, not the result of a value that could only be computed at runtime.
      * const THREE\_HOURS\_IN\_SECONDS: u32 = 60 \* 60 \* 3;
      * Rust’s naming convention for constants is to use all uppercase with underscores between words.
  + Shadowing: You can declare a new variable with the same name as a previous variable. Rustaceans say that the first variable is *shadowed* by the second, which means that the second variable is what the compiler will see when you use the name of the variable.
    - In effect, the second variable overshadows the first, taking any uses of the variable name to itself **until either it itself is shadowed or the scope ends**.
    - Shadowing is different from marking a variable as mut because we’ll get a compile-time error if we accidentally try to reassign to this variable without using the let keyword. By using let, we can perform a few transformations on a value but have the variable be immutable after those transformations have been completed.
  + Every value in Rust is of a certain *data type*, which tells Rust what kind of data is being specified so it knows how to work with that data.
  + Keep in mind that Rust is a *statically typed* language, which means that it must know the types of all variables at compile time.
  + The compiler can usually infer what type we want to use based on the value and how we use it but in cases when many types are possible (and the compiler cannot make a unique determination), we must add a type annotation, like this:  
    let guess: u32 = "42".parse().expect("Not a number!");
  + We’ll look at two data type subsets: scalar and compound.
    - Scalar types:
      * A *scalar* type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and character.



* + - * When you’re compiling in debug mode, Rust includes checks for integer overflow, but not in release mode.
      * Rust’s floating-point types are f32 and f64, which are 32 bits and 64 bits in size, respectively.
      * Integer division truncates toward zero to the nearest integer.
      * Booleans are one byte in size.
      * Rust’s char type is four bytes in size and represents a Unicode scalar value, which means it can represent a lot more than just ASCII. However, a “character” isn’t really a concept in Unicode, so your human intuition for what a “character” is may not match up with what a char is in Rust. We’ll discuss this topic in detail in [“Storing UTF-8 Encoded Text with Strings”](https://rust-book.cs.brown.edu/ch08-02-strings.html#storing-utf-8-encoded-text-with-strings) in Chapter 8.
    - Compound types:
      * *Compound types* can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.
      * A *tuple* is a general way of grouping together a number of values with a variety of types into one compound type. Tuples have a fixed length: once declared, they cannot grow or shrink in size.
        + We’ve added optional type annotations in this example: let tup: (i32, f64, u8) = (500, 6.4, 1);
      * To get the individual values out of a tuple, we can use pattern matching to destructure a tuple value. We can also access a tuple element directly by using a period (.) followed by the index of the value we want to access.
      * The tuple without any values has a special name, *unit*. This value and its corresponding type are both written () and represent an empty value or an empty return type. Expressions implicitly return the unit value if they don’t return any other value.
      * Another way to have a collection of multiple values is with an *array*. Unlike a tuple, every element of an array must have the same type. Unlike arrays in some other languages, arrays in Rust have a fixed length.
      * Arrays are useful when you want your data allocated on the stack, the same as the other types we have seen so far, rather than the heap (we will discuss the stack and the heap more in [Chapter 4](https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html#the-stack-and-the-heap)).
      * A *vector* is a collection type provided by the standard library that *is* allowed to grow or shrink in size because its contents live on the heap (not a primitive compound type). If you’re unsure whether to use an array or a vector, chances are you should use a vector.
      * let a = [1, 2, 3, 4, 5];
      * let a: [i32; 5] = [1, 2, 3, 4, 5]; //Specifying data type and size.
      * let a = [3; 5]; //Repeat 3, 5 times.
  + Rust code uses snake case as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words.
  + In function signatures, you *must* declare the type of each parameter.
  + Statements and Expressions:
    - Rust is an expression-based language.
    - Statements are instructions that perform some action and **do not return a value**.
    - Expressions evaluate to a resultant value and make up most of the rest of the code that you’ll write in Rust.
    - Calling a function is an expression. Calling a macro is an expression. A new scope block created with curly brackets is an expression, for example:

fn *main*() {

    let y = {

        let x = 3;

        x + 1

    };

*println!*("The value of y is: {y}");

}

* + - A statement can contain an expression. For example, 6 in ‘let y = 6;’ is an expression. Similarly, the block being assigned to y in the above code snippet is an expression.
    - Note that the x + 1 line doesn’t have a semicolon at the end, which is unlike most of the lines you’ve seen so far. Expressions do not include ending semicolons. If you add a semicolon to the end of an expression, you turn it into a statement, and it will then not return a value.
    - This is why we say that “Function bodies are made up of a series of statements optionally ending in an expression.”
  + In Rust functions, we don’t name return values, but we must declare their type after an arrow (->).

fn *five*() -> i32 {

    5

}

* + We can return early using the ‘return’ keyword too.
  + if expressions:
    - The condition *must* be a bool.
    - Using too many else if expressions can clutter your code, so if you have more than one, you might want to refactor your code. Chapter 6 describes a powerful Rust branching construct called match for these cases.
    - Because if is an expression, we can use it on the right side of a let statement to assign the outcome to a variable.

let number = if condition { 5 } else { 6 };

* + - This won’t work because variables must have a single type, and Rust needs to know at compile time what type the number variable is, definitively.
  + Rust has three kinds of loops: loop, while, and for.
    - One of the uses of a loop is to retry an operation you know might fail, such as checking whether a thread has completed its job. You might also need to pass the result of that operation out of the loop to the rest of your code. To do this, you can add the value you want returned after the break expression you use to stop the loop.
    - You can also return from inside a loop. While break only exits the current loop, return always exits the current function.
    - If you have loops within loops, break and continue apply to the innermost loop at that point. You can optionally specify a *loop label* on a loop that you can then use with break or continue to specify that those keywords apply to the labeled loop instead of the innermost loop. Loop labels must begin with a single quote.

fn *main*() {

    let mut count = 0;

*'counting\_up*: loop {

*println!*("count = {count}");

        let mut remaining = 10;

        loop {

*println!*("remaining = {remaining}");

            if remaining == 9 {

                break;

            }

            if count == 2 {

                break *'counting\_up*;

            }

            remaining -= 1;

        }

        count += 1;

    }

*println!*("End count = {count}");

}

* + - while loop to run a loop as long as a condition remains true.
    - The while loop can be used to iterate through the elements on a collection, however, a more concise and efficient way is to use a for loop (for element in array {…}).
    - for loops are most widely used. Even to run some code a specific number of times, most Rustaceans would use a for loop. The way to do that would be to use a Range, provided by the standard library, which generates all numbers in sequence starting from one number and ending before another number (for number in (1..4) {…}).
* Understanding Ownership:
  + Ownership is Rust’s most unique feature. It enables Rust to make memory safety guarantees without needing a garbage collector.
  + Some background: Interpreted programming languages only catch exceptions at run-time (since there is no ‘system’ to read the entire code before execution). On the other hand, compiled programming language analyze the whole source code to create the binary. This allows them to catch rule violations at compile time, some violations may be obvious e.g., if we use a variable before declaring it, but other violations not so much, e.g., when memory accesses are dependent on user input. But, what if we introduced rules that, when enforced, eliminate the entire possibility of these non-obvious violations occurring? That’s exactly what Rust does!
  + Ownership is a discipline for ensuring the **safety** of Rust programs. [**Safety is the Absence of Undefined Behavior**](https://rust-book.cs.brown.edu/ch04-01-what-is-ownership.html#safety-is-the-absence-of-undefined-behavior)
  + Rust does not allow you to interpret memory as an array of bytes. Rust provides a particular way to think about memory. Ownership is a discipline for safely using memory within that way of thinking. The rest of this chapter will explain the Rust model of memory.
    - Variables live in **frames**. A frame is a mapping from variables to values *for* a single scope, such as a function.
    - Frames are organized into a **stack** of currently-called-functions.
    - *Note:* this memory model does not fully describe how Rust actually works!
    - Pointers and pointee.
    - Allocating data on the heap returns a pointer. Heap is where data can live indefinitely (not tied to the lifetime of a scope).
    - Rust provides a construct called [Box](https://doc.rust-lang.org/std/boxed/index.html) for putting data on the heap. (let a = Box::new([0; 1\_000\_000]);)
    - When we copy a pointer, the original pointer gets *moved (explained soon)*.
    - Rust does not allow manual deallocation of heap memory. Instead, Rust *automatically* frees a box’s heap memory. Here is an *almost* correct description of Rust’s policy for freeing boxes:
      * **Box deallocation principle (almost correct):** If a variable is bound to a box, when Rust deallocates the variable’s frame, then Rust deallocates the box’s heap memory.
      * BUT this can cause issues too! If we had 2 pointers pointing to the same heap memory, Rust would try to free the box’s heap memory *twice* on behalf of both variables. That’s undefined behavior too!
      * To avoid this situation, we finally arrive at ownership. When a is bound to Box::new([0; 1\_000\_000]), we say that a **owns** the box. The statement let b = a **moves** ownership of the box from a to b.
      * **Box deallocation principle (fully correct):** If a variable owns a box, when Rust deallocates the variable’s frame, then Rust deallocates the box’s heap memory.
    - **Moved heap data principle:** if a variable x moves ownership of heap data to another variable y (in any possible code flow), then x cannot be used after the move.
    - (Last question of the last quiz of chapter 4: *Moving* is a shallow copy. https://github.com/cognitive-engineering-lab/rust-book/issues/256)