

Ferris the Rustacean

**Rust Info**

Rust compiler/debugger/intellisense > every other language

Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.

It wasn’t always so clear, but the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before.

Programmers who need to “dip down” into lower-level control can do so with Rust, without taking on the customary risk of crashes or security holes, and without having to learn the fine points of a fickle toolchain

Who is Rust for?

Teams of Developers, Students, Companies, Open Source Developers, People who value speed and stability

Rust also brings contemporary developer tools to the systems programming world:

* Cargo, the included dependency manager and build tool, makes adding, compiling, and managing dependencies painless and consistent across the Rust ecosystem.
* Rustfmt ensures a consistent coding style across developers.
* The Rust Language Server powers Integrated Development Environment (IDE) integration for code completion and inline error messages.

By using these and other tools in the Rust ecosystem, developers can be productive while writing systems-level code.

Rust is for students and those who are interested in learning about systems concepts. Using Rust, many people have learned about topics like operating systems development. The community is very welcoming and happy to answer student questions. Through efforts such as this book, the Rust teams want to make systems concepts more accessible to more people, especially those new to programming.

Hundreds of companies, large and small, use Rust in production for a variety of tasks. Those tasks include command line tools, web services, DevOps tooling, embedded devices, audio and video analysis and transcoding, cryptocurrencies, bioinformatics, search engines, Internet of Things applications, machine learning, and even major parts of the Firefox web browser.

Rust is for people who want to build the Rust programming language, community, developer tools, and libraries. We’d love to have you contribute to the Rust language.

Rust is for people who crave speed and stability in a language. By speed, we mean the speed of the programs that you can create with Rust and the speed at which Rust lets you write them.

**Make a new program**

Cmd

Cd into Rust

Cargo new \*project name\*

Cd into project, then into src

\*Type in rustc main.rs\*

Run main.exe

Code in the main.rs file

Type cargo run or main.exe in the src directory in cmd

**Installing Rust**

Add C++ extension in Visual Studio.

Go to rustling.org

Click Install at the top of the screen

Download the .exe, save and run

Windows might not like it but oh well,

Install with the default (1)

Copy Cargo bin directory (%USERPROFILExxxxxxx) (right click to copy) (close)

Windows search bar environment click Environment variables button in advanced tab

Click path, edit, new, then paste the copy. Apply and exit all that.

To check and see if that’s working Open cmd

Make new directory (mkdir Rust), cd into it

To use Rust project generator type in: cargo new hello\_world (makes package)

(show file explorer, added new folder and inside, there is a .gitignore, Cargo.toml (settings), src holds main.rs )

Cd into hello\_world then into source folder

Then type rustc main.rs (will compile program, generates exec)

Run it (main.exe)

**In VS Code**

Install rls in extensions

Install Code Runner

Add folder to project folder

Then you should be able to run it

**Ready to go!**

To obtain user input and then print the result as output. We need to bring in the io (input.output) library into the project. It comes from the standard library (also known as std (not the best name))

**Variables and Mutability**

By default, variables are immutable. This is one of many nudges Rust gives you to write your code in a way that takes advantage of the safety and easy concurrency that Rust offers. However, you still have the option to make your variables mutable.

When a variable is immutable, once a value is bound to a name, you can’t change that value

This example shows how the compiler helps you find errors in your programs. Even though compiler errors can be frustrating, they only mean your program isn’t safely doing what you want it to do yet; they do not mean that you’re not a good programmer! Experienced Rustaceans still get compiler errors.

In Rust, the compiler guarantees that when you state that a value won’t change, it really won’t change. That means that when you’re reading and writing code, you don’t have to keep track of how and where a value might change. Your code is thus easier to reason through.

you aren’t allowed to use mut with constants. Constants aren’t just immutable by default—they’re always immutable

You declare constants using the const keyword instead of the let keyword, and the type of the value must be annotated

Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about.

The last difference is that constants may be set only to a constant expression, not the result of a function call or any other value that could only be computed at runtime.

**Shadowing**

you can declare a new variable with the same name as a previous variable, and the new variable shadows the previous variable. Rustaceans say that the first variable is shadowed by the second, which means that the second variable’s value is what appears when the variable is used

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.

**Data Types**

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. We’ll look at two data type subsets: scalar and compound.

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. In cases when many types are possible, such as when we converted a String to a numeric type using parse we must add a type annotation

**Scalar**

A scalar type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters. You may recognize these from other programming languages

**Integer Types**

An integer is a number without a fractional component. This type declaration indicates that the value it’s associated with should be an unsigned integer (signed integer types start with i, instead of u) that takes up 32 bits of space. Each variant in the Signed and Unsigned columns (for example, i16) can be used to declare the type of an integer value.

| **Length** | **Signed** | **Unsigned** |
| --- | --- | --- |
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| 128-bit | i128 | u128 |
| arch | isize | usize |

Each variant can be either signed or unsigned and has an explicit size. Signed and unsigned refer to whether it’s possible for the number to be negative—in other words, whether the number needs to have a sign with it (signed) or whether it will only ever be positive and can therefore be represented without a sign (unsigned). It’s like writing numbers on paper: when the sign matters, a number is shown with a plus sign or a minus sign; however, when it’s safe to assume the number is positive, it’s shown with no sign.

Each signed variant can store numbers from -(2n - 1) to 2n - 1 - 1 inclusive, where n is the number of bits that variant uses. So an i8 can store numbers from -(27) to 27 - 1, which equals -128 to 127. Unsigned variants can store numbers from 0 to 2n - 1, so a u8 can store numbers from 0 to 28 - 1, which equals 0 to 255.

**Floating-Point Types**

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust’s floating-point types are f32 and f64, which are 32 bits and 64 bits in size, respectively. The default type is f64 because on modern CPUs it’s roughly the same speed as f32 but is capable of more precision.

Floating-point numbers are represented according to the IEEE-754 standard. The f32 type is a single-precision float, and f64 has double precision.

**Numeric Operations**

Rust supports the basic mathematical operations you’d expect for all of the number types: addition, subtraction, multiplication, division, and remainder.

**Boolean**

As in most other programming languages, a Boolean type in Rust has two possible values: true and false. Booleans are one byte in size. The Boolean type in Rust is specified using bool. As in most other programming languages, a Boolean type in Rust has two possible values: true and false. Booleans are one byte in size. The Boolean type in Rust is specified using bool.

**Character Type**

So far we’ve worked only with numbers, but Rust supports letters too. Rust’s char type is the language’s most primitive alphabetic type, and the following code shows one way to use it. (Note that char literals are specified with single quotes, as opposed to string literals, which use double quotes.)

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. Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces are all valid char values in Rust. Unicode Scalar Values range from U+0000 to U+D7FF and U+E000 to U+10FFFF inclusive. 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.

**Compound Types**

Compound types can group multiple values into one type. Rust has two primitive compound types: tuples and arrays.

**The Tuple Type**

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 create a tuple by writing a comma-separated list of values inside parentheses. Each position in the tuple has a type, and the types of the different values in the tuple don’t have to be the same

The variable tup binds to the entire tuple, because a tuple is considered a single compound element. To get the individual values out of a tuple, we can use pattern matching to destructure a tuple value

This program first creates a tuple and binds it to the variable tup. It then uses a pattern with let to take tup and turn it into three separate variables, x, y, and z. This is called destructuring, because it breaks the single tuple into three parts.

In addition to destructuring through pattern matching, we can access a tuple element directly by using a period (.) followed by the index of the value we want to access

**The Array Type**

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. Arrays in Rust are different from arrays in some other languages because arrays in Rust have a fixed length, like tuples.

In Rust, the values going into an array are written as a comma-separated list inside square brackets

Arrays are useful when you want your data allocated on the stack rather than the heap or when you want to ensure you always have a fixed number of elements. An array isn’t as flexible as the vector type, though. A vector is a similar collection type provided by the standard library that is allowed to grow or shrink in size. If you’re unsure whether to use an array or a vector, you should probably use a vector.

An example of when you might want to use an array rather than a vector is in a program that needs to know the names of the months of the year. It’s very unlikely that such a program will need to add or remove months, so you can use an array because you know it will always contain 12 elements.

You would write an array’s type by using square brackets, and within the brackets include the type of each element, a semicolon, and then the number of elements in the array

let a: [i32; 5] = [1, 2, 3, 4, 5];

Here, i32 is the type of each element. After the semicolon, the number 5 indicates the array contains five elements.

Writing an array’s type this way looks similar to an alternative syntax for initializing an array: if you want to create an array that contains the same value for each element, you can specify the initial value, followed by a semicolon, and then the length of the array in square brackets, as shown here:

let a = [3; 5];

The array named a will contain 5 elements that will all be set to the value 3 initially. This is the same as writing let a = [3, 3, 3, 3, 3]; but in a more concise way.

**Accessing Array Elements**

An array is a single chunk of memory allocated on the stack. You can access elements of an array using indexing

In this example, the variable named first will get the value 1, because that is the value at index [0] in the array. The variable named second will get the value 2 from index [1] in the array

**Functions**

Functions are pervasive in Rust code. You’ve already seen one of the most important functions in the language: the main function, which is the entry point of many programs. You’ve also seen the fn keyword, which allows you to declare new functions.

Rust code uses snake case as the conventional style for function and variable names. In snake case, all letters are lowercase and underscores separate words.

Function definitions in Rust start with fn and have a set of parentheses after the function name. The curly brackets tell the compiler where the function body begins and ends.

Rust doesn’t care where you define your functions, only that they’re defined somewhere.

In function signatures, you must declare the type of each parameter. This is a deliberate decision in Rust’s design: requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what you mean.

Function bodies are made up of a series of statements optionally ending in an expression. So far, we’ve only covered functions without an ending expression, but you have seen an expression as part of a statement. Because Rust is an expression-based language, this is an important distinction to understand

Function definitions are also statements; the entire preceding example is a statement in itself.

Statements do not return values. Therefore, you can’t assign a let statement to another variable

fn main() {

let x = (let y = 6);

}

\*run to show how great of a debugger Rust has\*

The 5 in five is the function’s return value, which is why the return type is i32. Let’s examine this in more detail. There are two important bits: first, the line let x = five(); shows that we’re using the return value of a function to initialize a variable.

Second, the five function has no parameters and defines the type of the return value, but the body of the function is a lonely 5 with no semicolon because it’s an expression whose value we want to return.

Running this code will print The value of x is: 6. But if we place a semicolon at the end of the line containing x + 1, changing it from an expression to a statement, we’ll get an error.

**Comments**

Same as pretty much every other coding languages, 2 forward slashes, //

**Control Flow**

Deciding whether or not to run some code depending on if a condition is true and deciding to run some code repeatedly while a condition is true are basic building blocks in most programming languages. The most common constructs that let you control the flow of execution of Rust code are if expressions and loops

If statements don’t need (), it will provide a warning.

Because if is an expression, we can use it on the right side of a let statement

**Loops**

It’s often useful to execute a block of code more than once. For this task, Rust provides several loops. A loop runs through the code inside the loop body to the end and then starts immediately back at the beginning. To experiment with loops, let’s make a new project called loops.

Rust has three kinds of loops: loop, while, and for.

The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop., without a break it will go forever (ctrl + C to stop)

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. However, you might need to pass the result of that operation 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; that value will be returned out of the loop so you can use it

let result = loop {

counter += 1;

if counter == 10 {

break counter \* 2;

}

};

Before the loop, we declare a variable named counter and initialize it to 0. Then we declare a variable named result to hold the value returned from the loop. On every iteration of the loop, we add 1 to the counter variable, and then check whether the counter is equal to 10. When it is, we use the break keyword with the value counter \* 2. After the loop, we use a semicolon to end the statement that assigns the value to result

It’s often useful for a program to evaluate a condition within a loop. While the condition is true, the loop runs. When the condition ceases to be true, the program calls break, stopping the loop. This loop type could be implemented using a combination of loop, if, else, and break

However, this pattern is so common that Rust has a built-in language construct for it, called a while loop. This construct eliminates a lot of nesting that would be necessary if you used loop, if, else, and break, and it’s clearer. While a condition holds true, the code runs; otherwise, it exits the loop.

You could use the while construct to loop over the elements of a collection, such as an array.

let a = [10, 20, 30, 40, 50];

let mut index = 0;

while index < 5 {

println!("the value is: {}", a[index]);

index += 1;

}

the code counts up through the elements in the array. It starts at index 0, and then loops until it reaches the final index in the array (that is, when index < 5 is no longer true). Running this code will print every element in the array

As a more concise alternative, you can use a for loop and execute some code for each item in a collection

for element in a.iter() {

println!("the value is: {}", element);

}

When we run this code, we’ll see the same output as before. More importantly, we’ve now increased the safety of the code and eliminated the chance of bugs that might result from going beyond the end of the array or not going far enough and missing some items.

Here’s what the countdown would look like using a for loop and another method we’ve not yet talked about, rev, to reverse the range

for number in (1..4).rev() {

println!("{}!", number);

}

println!("LIFTOFF!!!");

This code is a bit nicer, isn’t it?

You are now all Rustaceans

**RUST Sources**

https://doc.rust-lang.org/book/ch00-00-introduction.html

https://doc.rust-lang.org/rust-by-example/index.html

Install Rust: https://www.youtube.com/watch?v=SigbqtVQnyI