[Variable](#_Toc462761699)s and Mutability 2

[Shadowing 5](#_Toc462761700)

[Data Types 7](#_Toc462761701)

[Scalar Types 7](#_Toc462761702)

[Integer Types 8](#_Toc462761703)

[Floating-Point Types 9](#_Toc462761704)

[Numeric Operations 10](#_Toc462761705)

[The Boolean Type 11](#_Toc462761706)

[The Character Type 11](#_Toc462761707)

[Compound Types 11](#_Toc462761708)

[Grouping Values into Tuples 12](#_Toc462761709)

[Tuple Indexing 12](#_Toc462761710)

[Arrays 13](#_Toc462761711)

[Accessing Array Elements 14](#_Toc462761712)

[Invalid Array Element Access 14](#_Toc462761713)

[How Functions Work 15](#_Toc462761714)

[Function Arguments 16](#_Toc462761715)

[Function Bodies 18](#_Toc462761716)

[Statements and Expressions 18](#_Toc462761717)

[Functions with Return Values 20](#_Toc462761718)

[Comments 22](#_Toc462761719)

[Control Flow 23](#_Toc462761720)

[If Expressions 23](#_Toc462761721)

[Multiple Conditions with else if 26](#_Toc462761722)

[Using if in a Binding 27](#_Toc462761723)

[Repetition with Loops 29](#_Toc462761724)

[Repeating Code with loop 29](#_Toc462761725)

[Conditional Loops With while 30](#_Toc462761726)

[Looping Through a Collection with for 30](#_Toc462761727)

[Summary 33](#_Toc462761728)

Chapter 3

Common Programming Concepts

We’ll start this chapter by looking at concepts that appear in almost every programming language and learning how they work in Rust. Many programming languages have much in common at their core. None of the concepts presented in this chapter are unique to Rust, but we’ll discuss Rust’s particular syntax and conventions concerning these common concepts.

Specifically, you’ll learn about variables, basic types, functions, comments, and control flow. These foundations will be in every Rust program, and learning them early will give you a strong core to start from.

PROD: START BOX

Keywords

The Rust language has a set of keywords that have been reserved for use, much like other languages do. Keep in mind that you cannot use these words as names of variables or functions. Most of the keywords have special meanings, and you’ll be using them to do various tasks in your Rust programs; a few have no current functionality associated with them but have been reserved for functionality that might be added in Rust in the future. You can find a list of the keywords in Appendix XX.

PROD: END BOX

Variables and Mutability

As mentioned in Chapter 2, by default variables are immutable. This is one of many nudges in Rust that encourages 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 bindings mutable. Let’s explore how and why Rust encourages you to favor immutability, and why you might want to opt out.

Immutable variables mean that once a value is bound, you can’t change that value. To illustrate, let’s generate a new project called binding in your projects directory by using cargo new --bin bindings.

Then, in your new bindings directory, open src/main.rs and replace its code with the following:

Filename: src/main.rs

fn main() {

let x = 5;

println!("The value of x is: {}", x);

x = 6;

println!("The value of x is: {}", x);

}

Save and run the program using cargo run: you should receive an error message, as shown in this output:

$ cargo run

Compiling bindings v0.0.1 (file:///projects/bindings)

error: re-assignment of immutable variable `x` [--explain E0384]

--> src/main.rs:4:5

4 |> x = 6;

|> ^^^^^

note: prior assignment occurs here

--> src/main.rs:2:9

2 |> let x = 5;

|> ^

This example shows how the compiler helps you find an error in your program. 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.

PROD: START BOX

Extended Error Explanations

Let’s look at one particularly useful aspect of errors. Rust encourages you to seek further information on the kind of error you’ve received by showing you output like this:

error: re-assignment of immutable variable `x` [--explain E0384]

This error indicates that if you pass the –explain flag to rustc with the provided error code, you can see an extended explanation that will try to explain common causes of and solutions to that kind of error. Not every error has a longer explanation, but many do. Here’s a portion of the explanation for the E0384 error you just received:

$ rustc --explain E0384

This error occurs when an attempt is made to reassign an immutable variable.

For example:

```

fn main(){

let x = 3;

x = 5; // error, reassignment of immutable variable

}

```

These explanations are beneficial if you’re stuck on an error, so don’t hesitate to look up the error code. Use these helpful compiler errors to perfect your code.

PROD: END BOX

The error indicates that the cause of the error is reassignment of immutable variable, because you tried to assign a second value to the immutable x variable.

It’s important that you get compile-time errors when you attempt to change a value that you previously designated as immutable because this very situation can lead to bugs. If one part of your code operates on the assumption that a value will never change and another part of your code changes that value, it’s possible that the first part of the code won’t do what it was designed to do. This cause of bugs can be difficult to track down after the fact, especially when the second piece of code changes the value only sometimes.

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, which can make code easier to reason about.

But mutability can be very useful. Bindings are immutable only by default; you can make them mutable by adding mut in front of the variable name. In addition to allowing this value to change, it conveys intent to future readers of the code by indicating that other parts of the code will be changing this variable value.

For example, change src/main.rs to the following:

Filename: src/main.rs

fn main() {

let mut x = 5;

println!("The value of x is: {}", x);

x = 6;

println!("The value of x is: {}", x);

}

When we run this program, we get the following:

$ cargo run

Compiling bindings v0.1.0 (file:///projects/bindings)

Running `target/debug/bindings`

The value of x is: 5

The value of x is: 6

Using mut, we’re allowed to change the value that x binds to from 5 to 6. In some cases, you’ll want to make a binding mutable because it makes the code easier to understand than an implementation that only uses immutable bindings. In cases where you’re using large data structures, mutating an instance in place may be faster than copying and returning newly allocated instances. It all depends on the trade-offs you want to make in your situation.

Shadowing

As you saw in the guessing game tutorial in Chapter 2, you can declare new bindings with the same name as a previous binding, and the new binding shadows the previous binding. Rustaceans say that the first binding is shadowed by the second, which means that the second binding’s value is what you’ll see when you use the variable. You can shadow a binding by using the same binding’s name and repeating the use of the let keyword as follows:

Filename: src/main.rs

fn main() {

let x = 5;

let x = x + 1;

let x = x \* 2;

println!("The value of x is: {}", x);

}

This program first binds x to a value of 5. Then it shadows x by repeating let x =, taking the original value and adding 1 so the value of x is then 6. The third let statement also shadows x, taking the previous value and multiplying it by 2 to give x a final value of 12. When you run this program, it will output the following:

$ cargo run

Compiling bindings v0.1.0 (file:///projects/bindings)

Running `target/debug/bindings`

The value of x is: 12

This is different than marking a binding as mut, because unless we use the let keyword again, we’ll get a compile-time error if we accidentally try to reassign to this binding. We can perform a few transformations on a value but have the binding be immutable after those transformations have been completed.

The other difference between mut and shadowing is that because you’re effectively creating a new binding when you use the let keyword again, you can change the type of the value you’re binding to but reuse the same name. For example, say your program asks a user to show how many spaces they want between some text by inputting space characters, but you really want to store that input as a number:

let spaces = " ";

let spaces = spaces.len();

This construct is allowed because the first spaces binding is a string type, and the second spaces binding, which is a brand-new binding that happens to have the same name as the first one, is a number type. Shadowing thus spares you from having to come up with different names, like spaces\_str and spaces\_num; instead, you can reuse the simpler spaces name. However, if you try to use mut for this, as shown here:

let mut spaces = " ";

spaces = spaces.len();

you’ll get a compile-time error because you’re not allowed to mutate a  
binding’s type:

error: mismatched types [--explain E0308]

-->

|>

4 |> spaces = spaces.len();

|> ^^^^^^^^^^^^ expected &-ptr, found usize

note: expected type `&str`

note: found type `usize`

error: aborting due to previous error

Now that we’ve explored how variables work, let’s look at more data types of values that we can bind variables to.

Data Types

Every value in Rust is of a certain type, which tells Rust what kind of data is being specified so it knows how to work with that data. In this section, we’ll look at a number of types that are built into the language and split into two subsets: scalar and compound.

Throughout this section, keep in mind that Rust is a statically typed language, which means that it must know the types of all bindings at compile time. The compiler can usually infer what type you want to use based on the value and how you use it. In cases when many types are possible, such as when we converted a String to a numeric type using parse in Chapter 2, you must add a type annotation, like this:

let guess: u32 = "42".parse().unwrap();

If you don’t add the type annotation here, Rust will display the following error, which means the compiler needs more information from you to know which possible type you want to use:

error: unable to infer enough type information about `\_`; type annotations or

generic parameter binding required [--explain E0282]

-->

|>

3 |> let guess = "42".parse().unwrap();

|> ^^^^^

You’ll see different type annotations as we discuss the various data types.

Scalar Types

A scalar type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, booleans, and characters. You’ll likely recognize these from other programming languages, but let’s jump into how they work in Rust.

Integer Types

An integer is a number without a fractional component. We used one integer type earlier in this chapter, the i32 type. This type declaration indicates that the value it’s associated with should be a signed integer (hence the i, as opposed to a u for unsigned) for a 32-bit system. Table 3-1 shows the built-in integer types in Rust. Each variant in the Signed and Unsigned columns (for example, i32) can be used to declare the type of an integer value.

Table 3-1: Integer Types in Rust Each variant in the signed and unsigned columns (for example, i32) 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 |
| arch | isize | usize |

Each variant can be either signed or unsigned and has an explicit size. Signed and unsigned refers to whether it’s possible for the number to be negative or positive; 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. Signed numbers are stored using two’s complement representation (if you’re unsure what this is, you can search for it online; an explanation is outside the scope of this book).

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, 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.

Additionally, the isize and usize types depend on the kind of computer your program is running on: 64-bits if you’re on a 64-bit architecture and 32-bits if you’re on a 32-bit architecture.

You can write integer literals in any of the forms shown in Table 3-2. Note that all number literals except the byte literal allow a type suffix, such as 57u8, and \_ as a visual separator, such as 1\_000.

Table 3-2: Integer Literals in Rust

| Number Literals | Example |
| --- | --- |
| Decimal | 98\_222 |
| Hex | 0xff |
| Octal | 0o77 |
| Binary | 0b1111\_0000 |
| Byte (u8 only) | b’A’ |

So how do you know which type of integer to use? If you’re unsure, Rust’s defaults are generally good choices, and integer types default to i32: it’s generally the fastest, even on 64-bit systems. The primary situation in which you’d use isize or usize is when indexing some sort of collection.

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 it’s roughly the same speed as f32 but has a larger precision. It’s possible to use an f64 type on 32-bit systems, but it will be slower than using an f32 type on those systems. Most of the time, trading potential worse performance for better precision is a reasonable initial choice, and you should benchmark your code if you suspect floating-point size is a problem in your situation. Chapter XX shows you how to run benchmarks.

Here’s an example that shows floating-point numbers in action:

Filename: src/main.rs

fn main() {

let x = 2.0; // f64

let y: f32 = 3.0; // f32

}

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 usual basic mathematic operations you’d expect for all of the number types: addition, subtraction, multiplication, division, and modulo. The following code shows how you’d use each one in a let statement:

Filename: src/main.rs

fn main() {

// addition

let sum = 5 + 10;

// subtraction

let difference = 95.5 - 4.3;

// multiplication

let product = 4 \* 30;

// division

let quotient = 56.7 / 32.2;

// modulo

let remainder = 43 % 5;

}

Each expression in these statements uses a mathematical operator and evaluates to a single value, which is then bound to a variable.

The Boolean Type

As in most other programming languages, a boolean type in Rust has two possible values: true and false. The boolean type in Rust is specified using bool. For example:

Filename: src/main.rs

fn main() {

let t = true;

let f: bool = false; // with explicit type annotation

}

The main way to consume boolean values is through conditionals, such as an if statement. We’ll cover how if statements work in Rust in the “Control Flow” section on page XX.

Production: See cross-reference above.

The Character Type

So far you’ve only worked 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:

Filename: src/main.rs

fn main() {

let c = ‘z’;

let z = ‘ℤ’;

let heart\_eyed\_cat = ‘😻’;

}

Rust’s char type represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII. Accented letters, Chinese/Japanese/Korean ideographs, emoji, and zero width spaces are all valid char types 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. We’ll discuss this topic in detail in the “Strings” section in Chapter 8.

Compound Types

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

Grouping Values into Tuples

A tuple is a general way of grouping together some number of other values with distinct types into one compound type.

You create a tuple by writing a comma-separated list of values inside parentheses. Each position in the tuple has a distinct type, and the types of the different values in the tuple don’t have to be the same. We’ve added optional type annotations in this example:

Filename: src/main.rs

fn main() {

let tup: (i32, f64, u8) = (500, 6.4, 1);

}

Note that the single name tup binds to the entire tuple, emphasizing the fact that a tuple is considered a single compound element. To get the individual values out of a tuple, you can use pattern matching to destructure a tuple value, like this:

Filename: src/main.rs

fn main() {

let tup = (500, 6.4, 1);

let (x, y, z) = tup;

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

}

This program first creates a tuple and binds it to the name tup. It then uses a pattern with let to take tup and turn it into three separate bindings, x, y, and z. This is called destructuring, because it breaks the single tuple into three parts. Finally, the program prints the value of y, which is 6.4.

Tuple Indexing

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

Filename: src/main.rs

fn main() {

let x: (i32, f64, u8) = (500, 6.4, 1);

let five\_hundred = x.0;

let six\_point\_four = x.1;

let one = x.2;

}

This program creates a tuple, x, and then makes new bindings to each element by using their index. As with most programming languages, the first index in a tuple is 0.

Arrays

Another way to bind a name to 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 than arrays in some other languages because arrays in Rust have a fixed length: once declared, they cannot grow or shrink in size.

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

Filename: src/main.rs

fn main() {

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

}

Although arrays can be useful because they’re a primitive type and therefore can be very fast to use, they aren’t as flexible as the vector type. The vector type 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: Chapter 8 discusses vectors in more detail.

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

let months = ["January", "February", "March", "April", "May", "June", "July",

"August", "September", "October", "November", "December"];

Accessing Array Elements

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

Filename: src/main.rs

fn main() {

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

let first = a[0];

let second = a[1];

}

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

Invalid Array Element Access

What happens if you try to access an element of an array that is past the end of the array? Say you change the example to the following:

Filename: src/main.rs

fn main() {

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

let element = a[10];

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

}

Running this code using cargo run produces the following result:

$ cargo run

Compiling arrays v0.1.0 (file:///projects/arrays)

Running `target/debug/arrays`

thread ‘<main>‘ panicked at ‘index out of bounds: the len is 5 but the index is

10’, src/main.rs:4

note: Run with `RUST\_BACKTRACE=1` for a backtrace.

error: Process didn’t exit successfully: `target/debug/arrays` (exit code: 101)

The compilation didn’t produce any errors, but the program results in a runtime error and didn’t exit successfully. When you attempt to access an element using indexing, Rust will check that the index you’ve specified is less than the array length. If the index is greater than the length, Rust will “panic,” which is the term Rust uses when a program exits with an error.

This is the first example of Rust’s safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing. Chapter XX discusses more of Rust’s error handling.

How Functions Work

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. Here’s a program that contains an example function definition:

Filename: src/main.rs

fn main() {

println!("Hello, world!");

another\_function();

}

fn another\_function() {

println!("Another function.");

}

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

You can call any function you’ve defined by entering its name followed by a set of parentheses. Because another\_function is defined in the program, it can be called from inside the main function. Note that we defined another\_function after the main function in the source code; we could have defined it before as well. Rust doesn’t care where you define your functions, only that they’re defined somewhere.

Let’s start a new binary project named functions to explore functions further. Place the another\_function example in src/main.rs and run it. You should see the following output:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

Running `target/debug/functions`

Hello, world!

Another function.

The lines execute in the order in which they appear in the main function. First, the “Hello, world!” message prints, and then another\_function is called and its message is printed.

Function Arguments

Functions can also take arguments. The following rewritten version of another\_function shows what arguments look like in Rust:

Filename: src/main.rs

fn main() {

another\_function(5);

}

fn another\_function(x: i32) {

println!("The value of x is: {}", x);

}

Try running this program; you should get the following output:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

Running `target/debug/functions`

The value of x is: 5

In the declaration of another\_function, one argument named x is placed. The type of x is specified as i32. When 5 is passed to another\_function, the println! macro puts 5 where the pair of curly braces were in the format string.

In function signatures, you must declare the type. 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.

When you want a function to have multiple arguments, just separate them inside the function signature with commas, like this:

Filename: src/main.rs

fn main() {

another\_function(5, 6);

}

fn another\_function(x: i32, y: i32) {

println!("The value of x is: {}", x);

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

}

This example creates a function with two arguments, both of which are i32 types. If your function has multiple arguments, the arguments don’t need to be the same type, but they just happen to be in this example. The function then prints out the values of both of its arguments.

Let’s try running this code. Replace the program currently in your function project’s main.rs file with the preceding example, and run it using cargo run:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

Running `target/debug/functions`

The value of x is: 5

The value of y is: 6

Because 5 is passed as the x argument and 6 is passed as the y argument, the two strings are printed with these values.

Function Bodies

Function bodies are made up of a series of statements optionally ending in an expression. So far, you’ve only seen functions without an ending expression, but you have seen expressions as parts of statements. Because Rust is an expression-based language, this is an important distinction to understand. Other languages don’t have the same distinctions, so let’s look at what statements and expressions are, and how their differences affect the bodies of functions.

Statements and Expressions

You’ve already used statements and expressions. Statements are instructions that perform some action and do not return a value. Expressions evaluate to a resulting value. Let’s look at some examples.

Creating a variable and assigning a value to it with the let keyword is a statement. In this example, let y = 6; is a statement:

Filename: src/main.rs

fn main() {

let y = 6;

}

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 binding to another binding, as the following code tries to do:

Filename: src/main.rs

fn main() {

let x = (let y = 6);

}

When you run this program, you’ll get an error like this:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

error: expected expression, found statement (`let`)

--> src/main.rs:2:14

2 |> let x = (let y = 6);

|> ^^^

note: variable declaration using `let` is a statement

error: aborting due to previous error

error: Could not compile `functions`.

The let y = 6 statement does not return a value, so there isn’t anything for x to bind to. This is different than in other languages, such as C and Ruby where the assignment returns the value of the assignment. In those languages, you can write x = y = 6 and have both x and y have the value 6; that is not the case in Rust.

Expressions are code that evaluate to something and make up most of the rest of the code that you’ll write in Rust. Consider a simple math operation, such as 5 + 6, which

is an expression that evaluates to the value 11. Expressions can be part of statements: in the previous example that had the statement let y = 6;, 6 is an expression that evaluates to the value 6. Calling a function is an expression. Calling a macro is an expression. The block that we use to create new scopes, {}, is an expression, for example:

Filename: src/main.rs

fn main() {

let x = 5;

let y = {

let x = 3;

x + 1

};

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

}

The following expression shown in the code is a block that, in this case, evaluates to 4, and then gets bound to y as part of the let statement:

{

let x = 3;

x + 1

}

is a block that, in this case, evaluates to 4, and then gets bound to y as part of the let statement.Note that the line containing x + 1 does not have a semicolon at the end, unlike most of the lines you’ve seen so far. This is the most important distinction between expressions and statements to remember: statements end in semicolons, whereas expressions do not. If you add a semicolon to the end of an expression, you turn it into a statement, which will then not return a value. Keep this in mind as you explore function return values and expressions next.

Functions with Return Values

Functions can return values to the code that calls them. You don’t name return values, but you do declare their type after an arrow (->). In Rust, the “return value of the function” is synonymous with the “value of the final expression in the block of the body of a function.” Here’s an example of a function that returns a value:

Filename: src/main.rs

fn five() -> i32 {

5

}

fn main() {

let x = five();

println!("The value of x is: {}", x);

}

There are no function calls, macros, or even let statements in the five function—just the number 5 by itself. That’s a perfectly valid function in Rust. Note that the function’s return type is specified, too, as -> i32. Try running this code; the output should look like this:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

Running `target/debug/functions`

The value of x is: 5

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 binding. Because the function five returns a 5, that line is the same as the following:

let x = 5;

Second, the five function requires no arguments 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. Let’s look at another example:

Filename: src/main.rs

fn main() {

let x = plus\_one(5);

println!("The value of x is: {}", x);

}

fn plus\_one(x: i32) -> i32 {

x + 1

}

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

Filename: src/main.rs

fn main() {

let x = plus\_one(5);

println!("The value of x is: {}", x);

}

fn plus\_one(x: i32) -> i32 {

x + 1;

}

Running this code produces an error, as follows:

$ cargo run

Compiling functions v0.1.0 (file:///projects/functions)

error: not all control paths return a value [--explain E0269]

--> src/main.rs:7:1

7 |> fn plus\_one(x: i32) -> i32 {

|> ^

help: consider removing this semicolon:

--> src/main.rs:8:10

8 |> x + 1;

|> ^

error: aborting due to previous error

error: Could not compile `functions`.

The main error message, “not all control paths return a value,” reveals the core issue with this code. The definition of the function plus\_one says that it will return an i32, but statements don’t evaluate to a value. Therefore, nothing is returned, which contradicts the function definition and results in an error. In this output, Rust provides an option to rectify this issue: it suggests removing the semicolon, which would fix the error.

Comments

All programmers strive to make their code easy to understand, but sometimes extra explanation is warranted. In these cases, programmers leave notes, or comments, in their source code that the compiler will ignore but people reading the source code may find useful.

Here’s a simple comment:

// Hello, world.

In Rust, comments must start with two slashes and will extend until the end of the line. For comments that extend beyond a single line, you’ll need to include // on each line, like this:

// So we’re doing something complicated here, long enough that we need

// multiple lines of comments to do it! Whew! Hopefully, this comment will

// explain what’s going on.

Comments can also be placed at the end of code lines:

Filename: src/main.rs

fn main() {

let lucky\_number = 7; // I’m feeling lucky today.

}

But you’ll more often see them used in this format:

Filename: src/main.rs

fn main() {

// I’m feeling lucky today.

let lucky\_number = 7;

}

That’s all there is to comments. They’re not particularly complicated.

Control Flow

Deciding whether or not to run some code depending on if a condition is true or 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 Expressions

An if expression allows you to branch your code depending on conditions. You provide a condition and then state, “If this condition is met, run this block of code. If the condition is not met, do not run this block of code.”

Create a new project called branches in your projects directory to explore the if expression. In the src/main.rs file, input the following:

Filename: src/main.rs

fn main() {

let number = 3;

if number < 5 {

println!("condition was true");

} else {

println!("condition was false");

}

}

All if expressions start with the keyword if, which is followed by a condition. In this case, the condition checks whether or not the variable number has a value less than 5. The block of code we want to execute if the condition is true is placed immediately after the condition inside curly braces. These blocks are sometimes called arms. Optionally, you can also include an else expression, as was done here, to give the program an alternative block of code to execute should the condition evaluate to false. If you don’t provide an else expression and the condition is false, the program will just skip the if block and move on to the next bit of code.

Try running this code; you should see the following output:

$ cargo run

Compiling branches v0.1.0 (file:///projects/branches)

Running `target/debug/branches`

condition was true

Let’s try changing the value of number to a value that makes the condition false to see what happens:

let number = 7;

Run the program again, and look at the output:

$ cargo run

Compiling branches v0.1.0 (file:///projects/branches)

Running `target/debug/branches`

condition was false

It’s also worth noting that the condition in this code must be a bool. To see what happens if the condition isn’t a bool, try running the following code:

Filename: src/main.rs

fn main() {

let number = 3;

if number {

println!("number was three");

}

}

The if condition evaluates to a value of 3 this time, and Rust throws an error:

Compiling branches v0.1.0 (file:///projects/branches)

error: mismatched types [--explain E0308]

--> src/main.rs:4:8

4 |> if number {

|> ^^^^^^ expected bool, found integral variable

note: expected type `bool`

note: found type `\_`

error: aborting due to previous error

Could not compile `branches`.

The error indicates that Rust expected a bool but got an integer. Rust will not automatically try to convert non-boolean types to a boolean, unlike languages such as Ruby and JavaScript. You must be explicit and always provide if with a boolean as its condition. If you wanted your if code block to run only when a number is not equal to 0, for example, you would change the if expression to the following:

Filename: src/main.rs

fn main() {

let number = 3;

if number != 0 {

println!("number was something other than zero");

}

}

Running this code will print number was something other than zero.

Multiple Conditions with else if

You can have multiple conditions by combining if and else in an else if expression. For example:

Filename: src/main.rs

fn main() {

let number = 6;

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");

}

}

This program has four possible paths it can take. After running it, you should see the following output:

$ cargo run

Compiling branches v0.1.0 (file:///projects/branches)

Running `target/debug/branches`

number is divisible by 3

When this program executes, it checks each if expression in turn and executes the first body for which the condition holds true. Note that even though 6 is divisible by 2, we don’t see the output number is divisible by 2, nor do we see the number is not divisible by 4, 3, or 2 text from the else block. The reason is that Rust will only execute the block for the first true condition, and once it finds one, it won’t even check the rest.

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.

Using if in a Binding

The last detail you need to know about if is that it’s an expression. That means that you can use it on the right side of a let binding, for instance:

Filename: src/main.rs

fn main() {

let condition = true;

let number = if condition {

5

} else {

6

};

println!("The value of number is: {}", number);

}

The number variable will be bound to a value based on the outcome of the if expression. Run this code to see what happens:

$ cargo run

Compiling branches v0.1.0 (file:///projects/branches)

Running `target/debug/branches`

The value of number is: 5

Remember that blocks of code evaluate to the last expression in them, and numbers by themselves are also expressions. In this case, the value of the whole if expression depends on which block of code executes. This means the value that results from both arms of the if must be the same type; in the previous example, they were both i32 integers. But what happens if the types are mismatched, as in the following example?

Filename: src/main.rs

fn main() {

let condition = true;

let number = if condition {

5

} else {

"six"

};

println!("The value of number is: {}", number);

}

When we run this code, we’ll get an error. The if and else arms have value types that are incompatible, and Rust indicates exactly where to find the problem in the program:

Compiling branches v0.1.0 (file:///projects/branches)

error[E0308]: if and else have incompatible types

--> src/main.rs:4:18

|

4 | let number = if condition {

| ^ expected integral variable, found reference

|

= note: expected type `{integer}`

= note: found type `&’static str`

error: aborting due to previous error

error: Could not compile `branches`.

The expression in the if block evaluates to an integer, and the expression in the else block evaluates to a string. This won’t work because variables must have a single type. Rust needs to know at compile time what type the number binding is, definitively, so it can verify at compile time that its type is valid everywhere we use number. Rust wouldn’t be able to do that if the type of number was only determined at runtime; the compiler would be more complex and would make fewer guarantees about the code if it had to keep track of multiple hypothetical types for any variable.

Repetition with 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. Let’s try each one.

Repeating Code with loop

The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.

As an example, change the src/main.rs file in your loops directory to look like this:

Filename: src/main.rs

fn main() {

loop {

println!("again!");

}

}

When we run this program, we’ll see again! printed over and over continuously until we stop the program manually. Most terminals support a keyboard shortcut, ctrl-C, to halt a program that is stuck in a continual loop. Give it a try:

$ cargo run

Compiling loops v0.1.0 (file:///projects/loops)

Running `target/debug/loops`

again!

again!

again!

again!

^Cagain!

The symbol ^C represents where you pressed ctrl-C. You may or may not see the word “again!” printed after the ^C, depending on where the code was in the loop when it received the halt signal.

Fortunately, Rust provides another, more reliable way to break out of a loop. You can place the break keyword within the loop to tell the program when to stop executing the loop. Recall that we did this in the guessing game in Chapter 2 to exit the program when the user won the game by guessing the correct number.

Conditional Loops with while

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, you call break, stopping the loop. This loop type could be implemented using a combination of loop, if, else, and break; you could try that now in a program, if you’d like.

However, this pattern is so common that Rust has a more efficient language construct for it, and it’s called a while loop. The following example uses while: the program loops three times, counting down each time. Then, after the loop, it prints another message and exits:

Filename: src/main.rs

fn main() {

let mut number = 3;

while number != 0 {

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

number = number - 1;

}

println!("LIFTOFF!!!");

}

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.

Looping Through a Collection with for

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

Filename: src/main.rs

fn main() {

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

let mut index = 0;

while index < 5 {

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

index = index + 1;

}

}

Here, 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 out every element in the array:

$ cargo run

Compiling loops v0.1.0 (file:///projects/loops)

Running `target/debug/loops`

the value is: 10

the value is: 20

the value is: 30

the value is: 40

the value is: 50

All five array values appear in the terminal, as expected. Even though index will reach a value of 6 at some point, the loop stops executing before trying to fetch a sixth value from the array.

But this approach is error prone; we could cause the program to panic if the index length is incorrect. It’s also slow, because the compiler needs to perform the conditional check on every element on every iteration through the loop.

As a more efficient alternative, you can use a for loop and execute some code for each item in a collection. A for loop looks like this:

Filename: src/main.rs

fn main() {

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

for element in a.iter() {

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

}

}

When we run this code, we’ll see the same output as in the previous example. More important, 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.

For example, in the previous code that uses the while loop, if you removed an item from the a array but forgot to update the condition to while index < 4, the code would panic. Using the for loop, you don’t need to remember to change any other code if you changed the number of values in the array.

If you’re wondering about the iter code in this example, keep reading! Chapter XX covers method syntax generally, and Chapter XX covers iterators specifically.

The safety and conciseness of for loops make them the most commonly used loop construct in Rust. Even in situations in which you want to run some code a certain number of times, as in the countdown example that used a while loop, most Rustaceans would use a for loop. The way to do that would be to use a Range, which is a type provided by the standard library that generates all numbers in sequence starting from one number and ending before another number.

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:

Filename: src/main.rs

fn main() {

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

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

}

println!("LIFTOFF!!!");

}

This code is a bit nicer, isn’t it?

Summary

You made it! That was a sizable chapter: you learned about variables, scalar and  
compound data types, functions, comments, if expressions, and loops! If you want to practice with the concepts discussed in this chapter, try building programs to do the following:

Convert temperatures between Fahrenheit and Celsius.

Generate the nth Fibonacci number.

Print the lyrics to the Christmas carol “The Twelve Days of Christmas,” taking advantage of the repetition in the song.

When you’re ready to move on, we’ll talk about a concept in Rust that doesn’t commonly exist in other programming languages: ownership.