Common Programming Concepts

This chapter covers concepts that appear in almost every programming language and 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 them in the context of Rust and explain the conventions around using these 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.

> **Keywords**
>
> The Rust language has a set of *keywords* that are reserved for use by the language only, much as in other languages. 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 to Rust in the future. You can find a list of the keywords in Appendix A.

Variables and Mutability

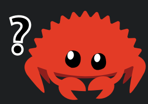
As mentioned in the “Storing Values with Variables” section, by default, variables are immutable. This is one of many nudges Rust gives you to write your code in a way that takes advantage of the safety and easy concurrency that Rust offers. However, you still have the option to make your variables mutable. Let’s explore how and why Rust encourages you to favor immutability and why sometimes you might want to opt out.

When a variable is immutable, once a value is bound to a name, you can’t change that value. To illustrate this, generate a new project called *variables* in your *projects* directory by using `cargo new variables`.

Then, in your new *variables* directory, open *src/main.rs* and replace its code with the following code, which won’t compile just yet:

Filename: src/main.rs

```rust
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 regarding an immutability error, as shown in this output:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 5;
  |         +++

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

This example shows how the compiler helps you find errors in your programs. Compiler errors can be frustrating, but really 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.

You received the error message `` cannot assign twice to immutable variable `x` `` because you tried to assign a second value to the immutable `x` variable.

It's important that we get compile-time errors when we attempt to change a value that's designated as immutable because this very situation can lead to bugs. If one part of our code operates on the assumption that a value will never change and another part of our code changes that value, it's possible that the first part of the code won't do what it was designed to do. The cause of this kind of bug can be difficult to track down after the fact, especially when the second piece of code changes the value only *sometimes*. The Rust compiler guarantees that when you state that a value won't change, it really won't change, so you don't have to keep track of it yourself. Your code is thus easier to reason through.

But mutability can be very useful, and can make code more convenient to write. Although variables are immutable by default, you can make them mutable by adding `mut` in front of the variable name as you did in Chapter 2. Adding `mut` also conveys intent to future readers of the code by indicating that other parts of the code will be changing this variable's value.

For example, let's 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 the program now, we get this:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

We're allowed to change the value bound to `x` from `5` to `6` when `mut` is used. Ultimately, deciding whether to use mutability or not is up to you and depends on what you think is clearest in that particular situation.

Constants

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.

First, 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. We'll cover types and type annotations in the next section, "Data Types", so don't worry about the details right now. Just know that you must always annotate the type.

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 value that could only be computed at runtime.

Here's an example of a constant declaration:

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

The constant's name is `THREE_HOURS_IN_SECONDS` and its value is set to the result of multiplying 60 (the number of seconds in a minute) by 60 (the number of minutes in an hour) by 3 (the number of hours we want to count in this program). Rust's naming convention for constants is to use all uppercase with underscores between words. The compiler is able to evaluate a limited set of operations at compile time, which lets us choose to write out this value in a way that's easier to understand and verify, rather than setting this constant to the value 10,800. See the Rust Reference's section on constant evaluation for more information on what operations can be used when declaring constants.

Constants are valid for the entire time a program runs, within the scope in which they were declared. This property makes constants useful for values in your application domain that multiple parts of the program might need to know about, such as the maximum number of points any player of a game is allowed to earn, or the speed of light.

Naming hardcoded values used throughout your program as constants is useful in conveying the meaning of that value to future maintainers of the code. It also helps to have only one place in your code you would need to change if the hardcoded value needed to be updated in the future.

Shadowing

As you saw in the guessing game tutorial in Chapter 2, 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. We can shadow a variable by using the same variable'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 in the inner scope is: {x}");
    }

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

This program first binds `x` to a value of `5`. Then it creates a new variable `x` by repeating `let x =`, taking the original value and adding `1` so the value of `x` is then `6`. Then, within an inner scope created with the curly brackets, the third `let` statement also shadows `x` and creates a new variable, multiplying the previous value by `2` to give `x` a value of `12`. When that scope is over, the inner shadowing ends and `x` returns to being `6`. When we run this program, it will output the following:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

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.

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

```
    let spaces = "   ";
    let spaces = spaces.len();
```

The first `spaces` variable is a string type and the second `spaces` variable is a number type. Shadowing thus spares us from having to come up with different names, such as `spaces_str` and `spaces_num`; instead, we can reuse the simpler `spaces` name. However, if we try to use `mut` for this, as shown here, we'll get a compile-time error:

```
    let mut spaces = "   ";
    spaces = spaces.len();
```

The error says we're not allowed to mutate a variable's type:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `variables` (bin "variables") due to 1 previous error
```

Now that we've explored how variables work, let's look at more data types they can have.

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` in the "Comparing the Guess to the Secret Number" section in Chapter 2, we must add a type annotation, like this:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

If we don't add the `: u32` type annotation shown in the preceding code, Rust will display the following error, which means the compiler needs more information from us to know which type we want to use:

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0284]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^        ----- type must be known at this point
  |
  = note: cannot satisfy `<_ as FromStr>::Err == _`
help: consider giving `guess` an explicit type
  |
2 |     let guess: /* Type */ = "42".parse().expect("Not a number!");
  |              ++++++++++++

For more information about this error, try `rustc --explain E0284`.
error: could not compile `no_type_annotations` (bin "no_type_annotations") due to 1
previous error
```

You'll see different type annotations for other 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 may recognize these from other programming languages. 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 in Chapter 2, the `u32` type. 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. Table 3-1 shows the built-in integer types in Rust. We can use any of these variants to declare the type of an integer value.

Table 3-1: Integer Types in Rust

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. Signed numbers are stored using two's complement representation.

Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive, where *n* is the number of bits that variant uses. So an `i8` can store numbers from $-(2^7)$ to $2^7 - 1$, which equals -128 to 127. Unsigned variants can store numbers from 0 to $2^n - 1$, so a `u8` can store numbers from 0 to $2^8 - 1$, which equals 0 to 255.

Additionally, the `isize` and `usize` types depend on the architecture of the computer your program is running on, which is denoted in the table as "arch": 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 number literals that can be multiple numeric types allow a type suffix, such as `57u8`, to designate the type. Number literals can also use `_` as a visual separator to make the number easier to read, such as `1_000`, which will have the same value as if you had specified `1000`.

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 places to start: integer types default to `i32`. The primary situation in which you'd use `isize` or `usize` is when indexing some sort of collection.

Integer Overflow

Let's say you have a variable of type `u8` that can hold values between 0 and 255. If you try to change the variable to a value outside that range, such as 256, *integer overflow* will occur, which can result in one of two behaviors. When you're compiling in debug mode, Rust includes checks for integer overflow that cause your program to *panic* at runtime if this behavior occurs. Rust uses the term *panicking* when a program exits with an error; we'll discuss panics in more depth in the "Unrecoverable Errors with `panic!`" section in Chapter 9.

When you're compiling in release mode with the `--release` flag, Rust does *not* include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs *two's complement wrapping*. In short, values greater than the maximum value the type can hold "wrap around" to the minimum of the values the type can hold. In the case of a `u8`, the value 256 becomes 0, the value 257 becomes 1, and so on. The program won't panic, but the variable will have a value that probably isn't what you were expecting it to have. Relying on integer overflow's wrapping behavior is considered an error.

To explicitly handle the possibility of overflow, you can use these families of methods provided by the standard library for primitive numeric types:

- Wrap in all modes with the `wrapping_*` methods, such as `wrapping_add`.
- Return the `None` value if there is overflow with the `checked_*` methods.
- Return the value and a boolean indicating whether there was overflow with the `overflowing_*` methods.
- Saturate at the value's minimum or maximum values with the `saturating_*` methods.

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. All floating-point types are signed.

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

Filename: src/main.rs

```rust
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 basic mathematical operations you'd expect for all the number types: addition, subtraction, multiplication, division, and remainder. Integer division truncates toward zero to the nearest integer. The following code shows how you'd use each numeric operation 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;
    let truncated = -5 / 3; // Results in -1

    // remainder
    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. Appendix B contains a list of all operators that Rust provides.

The Boolean Type

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`. For example:

Filename: src/main.rs

```
fn main() {
    let t = true;

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

The main way to use Boolean values is through conditionals, such as an `if` expression. We'll cover how `if` expressions work in Rust in the "Control Flow" section.

The Character Type

Rust's `char` type is the language's most primitive alphabetic type. Here are some examples of declaring `char` values:

Filename: src/main.rs

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Note that we specify `char` literals 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. We'll discuss this topic in detail in "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.

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. We've added optional type annotations in this example:

Filename: src/main.rs

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

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, 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 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. Finally, the program prints the value of `y`, which is `6.4`.

We can also access a tuple element directly by using a period (`.`) followed by the index of the value we 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 the tuple `x` and then accesses each element of the tuple using their respective indices. As with most programming languages, the first index in a tuple is 0.

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.

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. Unlike arrays in some other languages, arrays in Rust have a fixed length.

We write the values in an array as a comma-separated list inside square brackets:

Filename: src/main.rs

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

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) 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, chances are you should use a vector. Chapter 8 discusses vectors in more detail.

However, arrays are more useful when you know the number of elements will not need to change. For example, if you were using the names of the month in a program, you would probably use an array rather than a vector because you know it will always contain 12 elements:

```
let months = ["January", "February", "March", "April", "May", "June", "July",
              "August", "September", "October", "November", "December"];
```

You write an array's type using square brackets with the type of each element, a semicolon, and then the number of elements in the array, like so:

```
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.

You can also initialize an array to contain the same value for each element by specifying 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 of a known, fixed size that can be 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 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.

Invalid Array Element Access

Let's see what happens if you try to access an element of an array that is past the end of the array. Say you run this code, similar to the guessing game in Chapter 2, to get an array index from the user:

Filename: src/main.rs

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

This code compiles successfully. If you run this code using `cargo run` and enter `0`, `1`, `2`, `3`, or `4`, the program will print out the corresponding value at that index in the array. If you instead enter a number past the end of the array, such as `10`, you'll see output like this:

```
thread 'main' panicked at src/main.rs:19:19:
index out of bounds: the len is 5 but the index is 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

The program resulted in a *runtime* error at the point of using an invalid value in the indexing operation. The program exited with an error message and didn't execute the final `println!` statement. 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 or equal to the length, Rust will panic. This check has to happen at runtime, especially in this case, because the compiler can't possibly know what value a user will enter when they run the code later.

This is an example of Rust's memory 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 9 discusses more of Rust's error handling and how you can write readable, safe code that neither panics nor allows invalid memory access.

Functions

Functions are prevalent 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 which 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.");
}
```

We define a function in Rust by entering `fn` followed by a function name and a set of parentheses. The curly brackets tell the compiler where the function body begins and ends.

We can call any function we'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 in a scope that can be seen by the caller.

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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.28s
     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.

Parameters

We can define functions to have *parameters*, which are special variables that are part of a function's signature. When a function has parameters, you can provide it with concrete values for those parameters. Technically, the concrete values are called *arguments*, but in casual conversation, people tend to use the words *parameter* and *argument* interchangeably for either the variables in a function's definition or the concrete values passed in when you call a function.

In this version of `another_function` we add a parameter:

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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.21s
     Running `target/debug/functions`
The value of x is: 5
```

The declaration of `another_function` has one parameter named `x`. The type of `x` is specified as `i32`. When we pass `5` in to `another_function`, the `println!` macro puts `5` where the pair of curly brackets containing `x` was in the format string.

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 type you mean. The compiler is also able to give more helpful error messages if it knows what types the function expects.

When defining multiple parameters, separate the parameter declarations with commas, like this:

Filename: src/main.rs

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

This example creates a function named `print_labeled_measurement` with two parameters. The first parameter is named `value` and is an `i32`. The second is named `unit_label` and is type `char`. The function then prints text containing both the `value` and the `unit_label`.

Let's try running this code. Replace the program currently in your *functions* project's *src/main.rs* file with the preceding example and run it using `cargo run`:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

Because we called the function with `5` as the value for `value` and `'h'` as the value for `unit_label`, the program output contains those values.

Statements and Expressions

Function bodies are made up of a series of statements optionally ending in an expression. So far, the functions we've covered haven't included 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. 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** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value. Let's look at some examples.

We've actually already used statements and expressions. Creating a variable and assigning a value to it with the `let` keyword is a statement. In Listing 3-1, `let y = 6;` is a statement.

Filename: src/main.rs

```
fn main() {
    let y = 6;
}
```

Listing 3-1: A `main` function declaration containing one statement

Function definitions are also statements; the entire preceding example is a statement in itself. (As we will see below, *calling* a function is not a statement.)

Statements do not return values. Therefore, you can't assign a `let` statement to another variable, as the following code tries to do; you'll get an error:

Filename: src/main.rs

```
fn main() {
    let x = (let y = 6);
}
```

When you run this program, the error you'll get looks like this:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error: expected expression, found `let` statement
 --> src/main.rs:2:14
  |
2 |     let x = (let y = 6);
  |              ^^^
  |
  = note: only supported directly in conditions of `if` and `while` expressions

warning: unnecessary parentheses around assigned value
 --> src/main.rs:2:13
  |
2 |     let x = (let y = 6);
  |             ^         ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
2 -     let x = (let y = 6);
2 +     let x = let y = 6;
  |

warning: `functions` (bin "functions") generated 1 warning
error: could not compile `functions` (bin "functions") due to 1 previous error; 1
warning emitted
```

The `let y = 6` statement does not return a value, so there isn't anything for `x` to bind to. This is different from what happens 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 evaluate to a value and make up most of the rest of the code that you'll write in Rust. Consider a math operation, such as `5 + 6`, which is an expression that evaluates to the value `11`. Expressions can be part of statements: in Listing 3-1, the `6` in the statement `let y = 6;` is an expression that evaluates to the value `6`. 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:

Filename: src/main.rs

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

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

This expression:

```
{
    let x = 3;
    x + 1
}
```

is a block that, in this case, evaluates to `4`. That value gets bound to `y` as part of the `let` statement. 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. 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. We don't name return values, but we must 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. You can return early from a function by using the `return` keyword and specifying a value, but most functions return the last expression implicitly. 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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     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 variable. Because the function `five` returns a `5`, that line is the same as the following:

```
let x = 5;
```

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.

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`. 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:

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;
}
```

Compiling this code produces an error, as follows:

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: remove this semicolon to return this value

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` (bin "functions") due to 1 previous error
```

The main error message, `mismatched types`, 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, which is expressed by `()`, the unit type. Therefore, nothing is returned, which contradicts the function definition and results in an error. In this output, Rust provides a message to possibly help 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 *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, the idiomatic comment style starts a comment with two slashes, and the comment continues 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 lines containing code:

Filename: src/main.rs

```rust
fn main() {
    let lucky_number = 7; // I'm feeling lucky today
}
```

But you'll more often see them used in this format, with the comment on a separate line above the code it's annotating:

Filename: src/main.rs

```rust
fn main() {
    // I'm feeling lucky today
    let lucky_number = 7;
}
```

Rust also has another kind of comment, documentation comments, which we'll discuss in the "Publishing a Crate to Crates.io" section of Chapter 14.

Control Flow

The ability to run some code depending on whether a condition is `true` and 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`, followed by a condition. In this case, the condition checks whether or not the variable `number` has a value less than 5. We place the block of code to execute if the condition is `true` immediately after the condition inside curly brackets. Blocks of code associated with the conditions in `if` expressions are sometimes called *arms*, just like the arms in `match` expressions that we discussed in the "Comparing the Guess to the Secret Number" section of Chapter 2.

Optionally, we can also include an `else` expression, which we chose to do 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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     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:

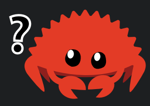
```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
condition was false
```

It's also worth noting that the condition in this code *must* be a `bool`. If the condition isn't a `bool`, we'll get an error. For example, 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:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

The error indicates that Rust expected a `bool` but got an integer. Unlike languages such as Ruby and JavaScript, Rust will not automatically try to convert non-Boolean types to a Boolean. You must be explicit and always provide `if` with a Boolean as its condition. If we want the `if` code block to run only when a number is not equal to `0`, for example, we can 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`.

Handling Multiple Conditions with `else if`

You can use 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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.31s
     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 evaluates to `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. That's because Rust only executes the block for the first `true` condition, and once it finds one, it doesn'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 `let` Statement

Because `if` is an expression, we can use it on the right side of a `let` statement to assign the outcome to a variable, as in Listing 3-2.

Filename: src/main.rs

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

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

Listing 3-2: Assigning the result of an `if` expression to a variable

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)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
The value of number is: 5
```

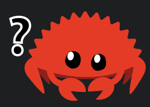
Remember that blocks of code evaluate to the last expression in them, and numbers by themselves are also expressions. In this case, the value of the whole `if` expression depends on which block of code executes. This means the values that have the potential to be results from each arm of the `if` must be the same type; in Listing 3-2, the results of both the `if` arm and the `else` arm were `i32` integers. If the types are mismatched, as in the following example, we'll get an error:

Filename: src/main.rs

```rust
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "six" };

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



When we try to compile 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:

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "six" };
  |                                 -          ^^^^^ expected integer, found `&str`
  |                                 |
  |                                 expected because of this

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` (bin "branches") due to 1 previous error
```

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, and Rust needs to know at compile time what type the `number` variable is, definitively. Knowing the type of `number` lets the compiler verify the 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*, which will run through the code inside the loop body to the end and then start 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 the keyboard shortcut `ctrl`-`c` to interrupt a program that is stuck in a continual loop. Give it a try:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     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 interrupt signal.

Fortunately, Rust also provides a way to break out of a loop using code. 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 the "Quitting After a Correct Guess" section of Chapter 2 to exit the program when the user won the game by guessing the correct number.

We also used `continue` in the guessing game, which in a loop tells the program to skip over any remaining code in this iteration of the loop and go to the next iteration.

Returning Values from Loops

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; that value will be returned out of the loop so you can use it, as shown here:

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

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`. Finally, we print the value in `result`, which in this case is `20`.

You can also `return` from inside a loop. While `break` only exits the current loop, `return` always exits the current function.

Loop Labels to Disambiguate Between Multiple Loops

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. Here's an example with two nested loops:

```rust
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}");
}
```

The outer loop has the label `'counting_up`, and it will count up from 0 to 2. The inner loop without a label counts down from 10 to 9. The first `break` that doesn't specify a label will exit the inner loop only. The `break 'counting_up;` statement will exit the outer loop. This code prints:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

Conditional Loops with `while`

A program will often need 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. It's possible to implement behavior like this 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 built-in language construct for it, called a `while` loop. In Listing 3-3, we use `while` to loop the program three times, counting down each time, and then, after the loop, print a message and exit.

Filename: src/main.rs

```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

Listing 3-3: Using a `while` loop to run code while a condition holds true

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 evaluates to `true`, the code runs; otherwise, it exits the loop.

Looping Through a Collection with `for`

You can also use the `while` construct to loop over the elements of a collection, such as an array. For example, the loop in Listing 3-4 prints each element in the array `a`.

Filename: src/main.rs

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

Listing 3-4: Looping through each element of a collection using a `while` loop

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 every element in the array:

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.32s
     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 `5` at some point, the loop stops executing before trying to fetch a sixth value from the array.

However, this approach is error prone; we could cause the program to panic if the index value or test condition is incorrect. For example, if you changed the definition of the `a` array to have four elements but forgot to update the condition to `while index < 4`, the code would panic. It's also slow, because the compiler adds runtime code to perform the conditional check of whether the index is within the bounds of the array on every iteration through the loop.

As a more concise alternative, you can use a `for` loop and execute some code for each item in a collection. A `for` loop looks like the code in Listing 3-5.

Filename: src/main.rs

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

Listing 3-5: Looping through each element of a collection using a `for` loop

When we run this code, we'll see the same output as in Listing 3-4. 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.

Using the `for` loop, you wouldn't need to remember to change any other code if you changed the number of values in the array, as you would with the method used in Listing 3-4.

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 in Listing 3-3, 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.

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! This was a sizable chapter: you learned about variables, scalar and compound data types, functions, comments, `if` expressions, and loops! To practice with the concepts discussed in this chapter, try building programs to do the following:

- Convert temperatures between Fahrenheit and Celsius.
- Generate the *n*th 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.