THE RUST PROGRAMMING LANGUAGE

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM THE RUST COMMUNITY

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *The Rust Programming Language* by Steve Klabnik and Carol Nichols, with contributions from the Rust Community! As a prepublication title, this book is probably incomplete, and it may have some rough edges.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

THE RUST PROGRAMMING LANGUAGE

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM
THE RUST COMMUNITY

Early Access edition, 4/7/17

ISBN-13: 978-1-59327-828-1

Publisher: William Pollock
Production Editor: Janelle Ludowise
Cover Illustration: Karen Rustad Tölva
Interior Design: Octopod Studios
Developmental Editor: Liz Chadwick
Technical Reviewer: Eddy Burt
Copyeditor: Anne Marie Walker
Compositors: Janelle Ludowise and Meg Sneeringer

BRIEF CONTENTS

2

GUESSING GAME

Let's jump into Rust by working through a hands-on project together! This chapter introduces you to a few common Rust concepts by showing you how to use them in a real program. You'll learn about `let`, `match`, methods, associated functions, using external crates, and more! The following chapters will explore these ideas in more detail. In this chapter, you'll practice the fundamentals.

We'll implement a classic beginner programming problem: a guessing game. Here's how it works: the program will generate a random integer between 1 and 100. It will then prompt the player to enter a guess. After entering a guess, it will indicate whether the guess is too low or too high. If the guess is correct, the game will print congratulations and exit.

Setting Up a New Project

To set up a new project, go to the *projects* directory that you created in Chapter 1, and make a new project using Cargo, like so:

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

The first command, `cargo new`, takes the name of the project (`guessing_game`) as the first argument. The `--bin` flag tells Cargo to make a binary project, similar to the one in Chapter 1. The second command changes to the new project's directory.

Look at the generated *Cargo.toml* file:

Filename: Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

If the author information that Cargo obtained from your environment is not correct, fix that in the file and save it again.

As you saw in Chapter 1, `cargo new` generates a "Hello, world!" program for you. Check out the *src/main.rs* file:

Filename: src/main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Now let's compile this "Hello, world!" program and run it in the same step using the `cargo run` command:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you need to rapidly iterate on a project, and this game is such a project: we want to quickly test each iteration before moving on to the next one.

Reopen the *src/main.rs* file. You'll be writing all the code in this file.

Processing a Guess

The first part of the program will ask for user input, process that input, and check that the input is in the expected form. To start, we'll allow the player to input a guess. Enter the code in Listing 2-1 into *src/main.rs*.

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

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

    println!("You guessed: {}", guess);
}
```

Listing 2-1: Code to get a guess from the user and print it out

This code contains a lot of information, so let's go over it bit by bit. To obtain user input and then print the result as output, we need to import the `io` (input/output) library from the standard library (which is known as `std`):

```
use std::io;
```

By default, Rust imports only a few types into every program in the *prelude*. If a type you want to use isn't in the prelude, you have to import that type into your program explicitly with a `use` statement. Using the `std::io` library provides you with a number of useful `io`-related features, including the functionality to accept user input.

As you saw in Chapter 1, the `main` function is the entry point into the program:

```
fn main() {
```

The `fn` syntax declares a new function, the `()` indicate there are no arguments, and `{` starts the body of the function.

As you also learned in Chapter 1, `println!` is a macro that prints a string to the screen:

```
println!("Guess the number!");

println!("Please input your guess.");
```

This code is just printing a prompt stating what the game is and requesting input from the user.

Storing Values with Variables

Next, we'll create a place to store the user input, like this:

```
let mut guess = String::new();
```

Now the program is getting interesting! There's a lot going on in this little line. Notice that this is a `let` statement, which is used to create *variables*. Here's another example:

```
let foo = bar;
```

This line will create a new variable named `foo` and bind it to the value `bar`. In Rust, variables are immutable by default. The following example shows how to use `mut` before the variable name to make a variable mutable:

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

NOTE *The `//` syntax starts a comment that continues until the end of the line. Rust ignores everything in comments.*

Now you know that `let mut guess` will introduce a mutable variable named `guess`. On the other side of the equal sign (`=`) is the value that `guess` is bound to, which is the result of calling `String::new`, a function that returns a new instance of a `String`. `String` is a string type provided by the standard library that is a growable, UTF-8 encoded bit of text.

The `::` syntax in the `::new` line indicates that `new` is an *associated function* of the `String` type. An associated function is implemented on a type, in this case `String`, rather than on a particular instance of a `String`. Some languages call this a *static method*.

This `new` function creates a new, empty `String`. You'll find a `new` function on many types, because it's a common name for a function that makes a new value of some kind.

To summarize, the `let mut guess = String::new();` line has created a mutable variable that is currently bound to a new, empty instance of a `String`. Whew!

Recall that we included the input/output functionality from the standard library with `use std::io;` on the first line of the program. Now we'll call an associated function, `stdin`, on `io`:

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

If we hadn't listed the `use std::io` line at the beginning of the program, we could have written this function call as `std::io::stdin`. The `stdin` function returns an instance of `std::io::Stdin`, which is a type that represents a handle to the standard input for your terminal.

The next part of the code, `.read_line(&mut guess)`, calls the `read_line` method on the standard input handle to get input from the user. We're also passing one argument to `read_line`: `&mut guess`.

The job of `read_line` is to take whatever the user types into standard input and place that into a string, so it takes that string as an argument. The string argument needs to be mutable so the method can change the string's content by adding the user input.

The `&` indicates that this argument is a *reference*, which gives you a way to let multiple parts of your code access one piece of data without needing to copy that data into memory multiple times. References are a complex feature, and one of Rust's major advantages is how safe and easy it is to use references. You don't need to know a lot of those details to finish this program: Chapter 4 will explain references more thoroughly. For now, all you need to know is that like variables, references are immutable by default. Hence, you need to write `&mut guess` rather than `&guess` to make it mutable.

We're not quite done with this line of code. Although it's a single line of text, it's only the first part of the single logical line of code. The second part is this method:

```
.expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, it's often wise to introduce a newline and other whitespace to help break up long lines. We could have written this code as:

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

However, one long line is difficult to read, so it's best to divide it: two lines for two method calls. Now let's discuss what this line does.

Handling Potential Failure with the Result Type

As mentioned earlier, `read_line` puts what the user types into the string we're passing it, but it also returns a value—in this case, an `io::Result`. Rust has a number of types named `Result` in its standard library: a generic `Result` as well as specific versions for submodules, such as `io::Result`.

The `Result` types are *enumerations*, often referred to as *enums*. An enumeration is a type that can have a fixed set of values, and those values are called the enum's *variants*. Chapter 6 will cover enums in more detail.

For `Result`, the variants are `Ok` or `Err`. `Ok` indicates the operation was successful, and inside the `Ok` variant is the successfully generated value. `Err` means the operation failed, and `Err` contains information about how or why the operation failed.

The purpose of these `Result` types is to encode error handling information. Values of the `Result` type, like any type, have methods defined on them. An instance of `io::Result` has an `expect` method that you can call. If this instance of `io::Result` is an `Err` value, `expect` will cause the program to crash and display the message that you passed as an argument to `expect`. If the `read_line` method returns an `Err`, it would likely be the result of an error coming from the underlying operating system. If this instance of `io::Result`

is an `Ok` value, `expect` will take the return value that `Ok` is holding and return just that value to you so you can use it. In this case, that value is the number of characters the user entered into standard input.

If you don't call `expect`, the program will compile, but you'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust warns that you haven't used the `Result` value returned from `read_line`, indicating that the program hasn't handled a possible error.

The right way to suppress the warning is to actually write error handling, but since you just want to crash this program when a problem occurs, you can use `expect`. You'll learn about recovering from errors in Chapter 9.

Printing Values with println! Placeholders

Aside from the closing curly brackets, there's only one more line to discuss in the code added so far, which is the following:

```
println!("You guessed: {}", guess);
```

This line prints out the string we saved the user's input in. The set of {} is a placeholder: think of {} as little crab pincers that hold a value in place. You can print more than one value using {}: the first set of {} holds the first value listed after the format string, the second set holds the second value, and so on. Printing out multiple values in one call to `println!` would look like this:

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

This code would print out `x = 5 and y = 10`.

Testing the First Part

Let's test the first part of the guessing game. You can run it using **`cargo run`**:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

At this point, the first part of the game is done: we're getting input from the keyboard and then printing it.

Generating a Secret Number

Next, we need to generate a secret number that the user will try to guess. The secret number should be different every time so the game is fun to play more than once. Let's use a random number between 1 and 100 so the game isn't too difficult. Rust doesn't yet include random number functionality in its standard library. However, the Rust team does provide a `rand` crate at *https://crates.io/crates/rand*.

Using a Crate to Get More Functionality

Remember that a *crate* is a package of Rust code. The project we've been building is a *binary crate*, which is an executable. The `rand` crate is a *library crate*, which contains code intended to be used in other programs.

Cargo's use of external crates is where it really shines. Before we can write code that uses `rand`, we need to modify the *Cargo.toml* file to include the `rand` crate as a dependency. Open that file now and add the following line to the bottom beneath the `[dependencies]` section header that Cargo created for you:

Filename: Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

In the *Cargo.toml* file, everything that follows a header is part of a section that continues until another section starts. The `[dependencies]` section is where you tell Cargo which external crates your project depends on and which versions of those crates you require. In this case, we'll specify the `rand` crate with the semantic version specifier `0.3.14`. Cargo understands Semantic Versioning (sometimes called *SemVer*), which is a standard for writing version numbers. The number `0.3.14` is actually shorthand for `^0.3.14`, which means "any version that has a public API compatible with version 0.3.14."

Now, without changing any of the code, let's build the project, as shown in Listing 2-2:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

Listing 2-2: The output from running `cargo build` after adding the `rand` crate as a dependency

You may see different version numbers (but they will all be compatible with the code, thanks to SemVer!), and the lines may be in a different order.

Now that we have an external dependency, Cargo fetches the latest versions of everything from the *registry*, which is a copy of data from https://crates.io. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks the `[dependencies]` section and downloads any you don't have yet. In this case, although we only listed `rand` as a dependency, Cargo also grabbed a copy of `libc`, because `rand` depends on `libc` to work. After downloading them, Rust compiles them and then compiles the project with the dependencies available.

If you immediately run `cargo build` again without making any changes, you won't get any output. Cargo knows it has already downloaded and compiled the dependencies, and you haven't changed anything about them in your *Cargo.toml* file. Cargo also knows that you haven't changed anything about your code, so it doesn't recompile that either. With nothing to do, it simply exits. If you open up the *src/main.rs* file, make a trivial change, and then save it and build again, you'll only see one line of output:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

This line shows Cargo only updates the build with your tiny change to the *src/main.rs* file. Your dependencies haven't changed, so Cargo knows it can reuse what it has already downloaded and compiled for those. It just rebuilds your part of the code.

The Cargo.lock File Ensures Reproducible Builds

Cargo has a mechanism that ensures you can rebuild the same artifact every time you or anyone else builds your code: Cargo will use only the versions of the dependencies you specified until you indicate otherwise. For example, what happens if next week version `v0.3.15` of the `rand` crate comes out and contains an important bug fix but also contains a regression that will break your code?

The answer to this problem is the *Cargo.lock* file, which was created the first time you ran `cargo build` and is now in your *guessing_game* directory. When you build a project for the first time, Cargo figures out all the versions of the dependencies that fit the criteria and then writes them to the *Cargo.lock* file. When you build your project in the future, Cargo will see that the *Cargo.lock* file exists and use the versions specified there rather than doing all the work of figuring out versions again. This lets you have a reproducible build automatically. In other words, your project will remain at `0.3.14` until you explicitly upgrade, thanks to the *Cargo.lock* file.

Updating a Crate to Get a New Version

When you *do* want to update a crate, Cargo provides another command, `update`, which will:

1. Ignore the *Cargo.lock* file and figure out all the latest versions that fit your specifications in *Cargo.toml*.
2. If that works, Cargo will write those versions to the *Cargo.lock* file.

But by default, Cargo will only look for versions larger than `0.3.0` and smaller than `0.4.0`. If the `rand` crate has released two new versions, `0.3.15` and `0.4.0`, you would see the following if you ran `cargo update`:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3.15
```

At this point, you would also notice a change in your *Cargo.lock* file noting that the version of the `rand` crate you are now using is `0.3.15`.

If you wanted to use `rand` version `0.4.0` or any version in the `0.4.x` series, you'd have to update the *Cargo.toml* file to look like this instead:

```
[dependencies]

rand = "0.4.0"
```

The next time you run `cargo build`, Cargo will update the registry of crates available and reevaluate your `rand` requirements according to the new version you specified.

There's a lot more to say about Cargo and its ecosystem that we'll discuss in Chapter XX, but for now, that's all you need to know. Cargo makes it very easy to reuse libraries, so Rustaceans are able to write smaller projects that are assembled from a number of packages.

Generating a Random Number

Let's start *using* `rand`. The next step is to update *src/main.rs*, as shown in Listing 2-3:

Filename: src/main.rs

```
❶ extern crate rand;

use std::io;
❷ use rand::Rng;

fn main() {
    println!("Guess the number!");
```

```
❸     let secret_number = rand::thread_rng().gen_range(1, 101);

      println!("The secret number is: {}", secret_number);

      println!("Please input your guess.");

      let mut guess = String::new();

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

      println!("You guessed: {}", guess);
  }
```

Listing 2-3: Code changes needed in order to generate a random number

First, we add a line to the top ❶ that lets Rust know we'll be using that external dependency. This also does the equivalent of calling `use rand`, so now we can call anything in the `rand` crate by prefixing it with `rand::`.

Next, we add another `use` line: `use rand::Rng` ❷. `Rng` is a trait that defines methods that random number generators implement, and this trait must be in scope for us to use those methods. Chapter 10 will cover traits in detail.

Also, we're adding two more lines in the middle ❸. The `rand::thread_rng` function will give us the particular random number generator that we're going to use: one that is local to the current thread of execution and seeded by the operating system. Next, we call the `gen_range` method on the random number generator. This method is defined by the `Rng` trait that we brought into scope with the `use rand::Rng` statement. The `gen_range` method takes two numbers as arguments and generates a random number between them. It's inclusive on the lower bound but exclusive on the upper bound, so we need to specify `1` and `101` to request a number between 1 and 100.

Knowing which traits to import and which functions and methods to use from a crate isn't something that you'll just *know*. Instructions for using a crate are in each crate's documentation. Another neat feature of Cargo is that you can run the `cargo doc --open` command, which will build documentation provided by all of your dependencies locally and open it in your browser. If you're interested in other functionality in the `rand` crate, for example, run `cargo doc --open` and click **rand** in the sidebar on the left.

The second line that we added to the code prints the secret number. This is useful while we're developing the program to be able to test it, but we'll delete it from the final version. It's not much of a game if the program prints the answer as soon as it starts!

Try running the program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
```

```
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

You should get different random numbers, and they should all be numbers between 1 and 100. Great job!

Comparing the Guess to the Secret Number

Now that we have user input and a random number, we can compare them. That step is shown in Listing 2-4:

```
extern crate rand;

use std::io;
❶ use std::cmp::Ordering;
use rand::Rng;

fn main() {

---snip---

    println!("You guessed: {}", guess);

    match❷ guess.cmp(&secret_number)❸ {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

Listing 2-4: Handling the possible return values of comparing two numbers

The first new bit here is another `use` ❶, bringing a type called `std::cmp::Ordering` into scope from the standard library. `Ordering` is another enum, like `Result`, but the variants for `Ordering` are `Less`, `Greater`, and `Equal`. These are the three outcomes that are possible when you compare two values.

Then we add five new lines at the bottom that use the `Ordering` type. The `cmp` method ❸ compares two values and can be called on anything that can be compared. It takes a reference to whatever you want to compare with: here it's comparing the `guess` to the `secret_number`. `cmp` returns a variant of the `Ordering` enum we imported with the `use` statement. We use

a `match` expression ❷ to decide what to do next based on which variant of `Ordering` was returned from the call to `cmp` with the values in `guess` and `secret_number`.

A `match` expression is made up of *arms*. An arm consists of a *pattern* and the code that should be run if the value given to the beginning of the `match` expression fits that arm's pattern. Rust takes the value given to `match` and looks through each arm's pattern in turn. The `match` construct and patterns are powerful features in Rust that let you express a variety of situations your code might encounter and help ensure that you handle them all. These features will be covered in detail in Chapter 6 and Chapter XX, respectively.

Let's walk through an example of what would happen with the `match` expression used here. Say that the user has guessed 50, and the randomly generated secret number this time is 38. When the code compares 50 to 38, the `cmp` method will return `Ordering::Greater`, because 50 is greater than 38. `Ordering::Greater` is the value that the `match` expression gets. It looks at the first arm's pattern, `Ordering::Less`, and sees that the value `Ordering::Greater` does not match `Ordering::Less`. So it ignores the code in that arm and moves to the next arm. The next arm's pattern, `Ordering::Greater`, *does* match `Ordering::Greater`! The associated code in that arm will execute and print `Too big!` to the screen. The `match` expression ends because it has no need to look at the last arm in this particular scenario.

However, the code in Listing 2-4 won't compile yet. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ expected struct `std::string::String`,
found integral variable
   |
   = note: expected type `&std::string::String`
   = note:    found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

The core of the error states that there are *mismatched types*. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess = String::new()`, Rust was able to infer that `guess` should be a `String` and didn't make us write the type. The `secret_number`, on the other hand, is a number type. A few number types can have a value between 1 and 100: `i32`, a 32-bit number; `u32`, an unsigned 32-bit number; `i64`, a 64-bit number; as well as others. Rust defaults to an `i32`, which is the type of `secret_number` unless you add type information elsewhere that would cause Rust to infer a different numerical type. The reason for the error is that Rust will not compare a string and a number type.

Ultimately, we want to convert the `String` the program reads as input into a real number type so we can compare it to the guess numerically. We can do that by adding the following two lines to the `main` function body:

```
---snip---

    let mut guess = String::new();

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

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

We create a variable named `guess`. But wait, doesn't the program already have a variable named `guess`? It does, but Rust allows us to *shadow* the previous value of `guess` with a new one. This feature is often used in similar situations in which you want to convert a value from one type to another type. Shadowing lets us reuse the `guess` variable name rather than forcing us to create two unique variables, like `guess_str` and `guess` for example. (Chapter 3 covers shadowing in more detail.)

We bind `guess` to the expression `guess.trim().parse()`. The `guess` in the expression refers to the original `guess` that was a `String` with the input in it. The `trim` method on a `String` instance will eliminate any whitespace at the beginning and end. `u32` can only contain numerical characters, but the user must press the ENTER key to satisfy `read_line`. When the user presses ENTER, a newline character is added to the string. For example, if the user types `5` and presses ENTER, `guess` looks like this: `5\n`. The `\n` represents "newline," the return key. The `trim` method eliminates `\n`, resulting in just `5`.

The `parse` method on strings parses a string into some kind of number. Because this method can parse a variety of number types, we need to tell Rust the exact number type we want by using `let guess: u32`. The colon (`:`) after `guess` tells Rust we'll annotate the variable's type. Rust has a few built-in number types; the `u32` seen here is an unsigned, 32-bit integer. It's a good default choice for a small positive number. You'll learn about other number types in Chapter 3. Additionally, the `u32` annotation in this example program and the comparison with `secret_number` means that Rust will infer that `secret_number` should be a `u32` as well. So now the comparison will be between two values of the same type!

The call to `parse` could easily cause an error. If, for example, the string contained `A👍%`, there would be no way to convert that to a number. Because it might fail, the `parse` method returns a `Result` type, much like the `read_line` method does as discussed earlier in "Handling Potential Failure with the Result Type" on page XX. We'll treat this `Result` the same way by using the `expect` method again. If `parse` returns an `Err Result` variant because it couldn't create a number from the string, the `expect` call will crash the game and print the message we give it. If `parse` can successfully convert the string to a number, it will return the `Ok` variant of `Result`, and `expect` will return the number that we want from the `Ok` value.

Let's run the program now!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! Even though spaces were added before the guess, the program still figured out that the user guessed 76. Run the program a few times to verify the different behavior with different kinds of input: guess the number correctly, guess a number that is too high, and guess a number that is too low.

We have most of the game working now, but the user can make only one guess. Let's change that by adding a loop!

Allowing Multiple Guesses with Looping

The `loop` keyword gives us an infinite loop. We'll add that now to give users more chances at guessing the number:

```
---snip---

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        ---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
    }
}
```

As you can see, we've moved everything into a loop from the guess input prompt onward. Be sure to indent those lines another four spaces each, and run the program again. Notice that there is a new problem because the program is doing exactly what we told it to do: ask for another guess forever! It doesn't seem like the user can quit!

The user could always halt the program by using the keyboard shortcut CTRL-C. But there's another way to escape this insatiable monster, as mentioned in the `parse` discussion in "Comparing the Guesses" on page XX: if the user enters a non-number answer, the program will crash. The user can take advantage of that in order to quit, as shown here:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)
```

Typing `quit` actually quits the game, but so will any other non-number input. However, this is suboptimal to say the least. We want the game to automatically stop when the correct number is guessed.

Quitting After a Correct Guess

Let's program the game to quit when the user wins by adding a `break`:

```
---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
```

```
            }
        }
    }
}
```

By adding the `break` line after `You win!`, the program will exit the loop when the user guesses the secret number correctly. Exiting the loop also means exiting the program, because the loop is the last part of `main`.

Handling Invalid Input

To further refine the game's behavior, rather than crashing the program when the user inputs a non-number, let's make the game ignore a non-number so the user can continue guessing. We can do that by altering the line where `guess` is converted from a `String` to a `u32`:

```
---snip---

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

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("You guessed: {}", guess);

---snip---
```

Switching from an `expect` call to a `match` expression is how you generally move from crash on error to actually handling the error. Remember that `parse` returns a `Result` type, and `Result` is an enum that has the variants `Ok` or `Err`. We're using a `match` expression here, like we did with the `Ordering` result of the `cmp` method.

If `parse` is able to successfully turn the string into a number, it will return an `Ok` value that contains the resulting number. That `Ok` value will match the first arm's pattern, and the `match` expression will just return the `num` value that `parse` produced and put inside the `Ok` value. That number will end up right where we want it in the new `guess` variable we're creating.

If `parse` is *not* able to turn the string into a number, it will return an `Err` value that contains more information about the error. The `Err` value does not match the `Ok(num)` pattern in the first `match` arm, but it does match the `Err(_)` pattern in the second arm. The `_` is a catchall value; in this example, we're saying we want to match all `Err` values, no matter what information they have inside them. So the program will execute the second arm's code, `continue`, which means to go to the next iteration of the `loop` and ask for another guess. So effectively, the program ignores all errors that `parse` might encounter!

Now everything in the program should work as expected. Let's try it by running `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny final tweak, we will finish the guessing game: recall that the program is still printing out the secret number. That worked well for testing, but it ruins the game. Let's delete the `println!` that outputs the secret number. Listing 2-5 shows the final code:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

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

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);
```

```
        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!(“You win!”);
                break;
            }
        }
    }
}
```

Listing 2-5: Complete code of the guessing game

Summary

At this point, you've successfully built the guessing game! Congratulations!

This project was a hands-on way to introduce you to many new Rust concepts: `let`, `match`, methods, associated functions, using external crates, and more. In the next few chapters, you'll learn about these concepts in more detail. Chapter 3 covers concepts that most programming languages have, such as variables, data types, and functions, and shows how to use them in Rust. Chapter 4 explores ownership, which is a Rust feature that is most different from other languages. Chapter 5 discusses structs and method syntax, and Chapter 6 endeavors to explain enums.

3

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 their conventions.

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 have been reserved for use by the language only, 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 to Rust in the future. You can find a list of the keywords in Appendix A.

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 variables mutable. Let's explore how and why Rust encourages you to favor immutability, and why you might want to opt out.

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

Then, in your new *variables* 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:

```
error[E0384]: re-assignment of 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;
  |     ^^^^^ re-assignment of immutable variable
```

This example shows how the compiler helps you find errors in your programs. Even though compiler errors can be frustrating, they only mean

your program isn't safely doing what you want it to do yet; they do *not* mean that you're not a good programmer! Experienced Rustaceans still get compiler errors

The error indicates that the cause of the error is `re-assignment of immutable variable`, because we 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 we previously 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. 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. Variables 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, let's change *src/main.rs* to the following:

```
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 variables v0.1.0 (file:///projects/variables)
     Running `target/debug/variables`
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 variable mutable because it makes the code more convenient to write than an implementation that only uses immutable variables.

There are multiple trade-offs to consider in addition to the prevention of bugs. For example, in cases where you're using large data structures, mutating an instance in place may be faster than copying and returning newly allocated instances. With smaller data structures,

creating new instances and writing in a more functional programming style may be easier to reason about, so the lower performance might be a worthwhile penalty for gaining that clarity.

Differences Between Variables and Constants

Being unable to change the value of a variable might have reminded you of another programming concept that most other languages have: *constants*. Like immutable variables, constants are also 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 only 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're about to 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 the code need to know about.

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

Here's an example of a constant declaration where the constant's name is `MAX_POINTS` and its value is set to 100,000. (Rust constant naming convention is to use all upper case with underscores between words):

```
const MAX_POINTS: u32 = 100_000;
```

Constants are valid for the entire time a program runs, within the scope they were declared in, making them a useful choice 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, we can declare new variables with the same name as previous variables, and the new variable *shadows* the previous variable. Rustaceans say that the first variable is

shadowed by the second, which means that the second variable's value is what we'll see when we use the variable. We can shadow a variable by using the same variable's name and repeating the use of the `let` keyword as follows:

```
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 we run this program, it will output the following:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
     Running `target/debug/variables`
The value of x is: 12
```

This is different than marking a variable 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 variable. 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, but we really want to store that input as a number:

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

This construct is allowed because the first `spaces` variable is a string type, and the second `spaces` variable, which is a brand-new variable that happens to have the same name as the first one, is a number type. Shadowing thus spares us from having to come up with different names, like `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:

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

we'll get a compile-time error because we're not allowed to mutate a variable's type:

```
error[E0308]: mismatched types
 --> src/main.rs:3:14
  |
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected &str, found usize
  |
  = note: expected type `&str`
             found type `usize`
```

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 *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. We split the types 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 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 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 type annotation here, Rust will display the following error, which means the compiler needs more information from us to know which possible type we want to use:

```
error[E0282]: unable to infer enough type information about `_`
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^ cannot infer type for `_`
  |
  = note: type annotations or generic parameter binding required
```

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) that takes up 32 bits of space. 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

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 - 1, which equals -128 to 127. Unsigned variants can store numbers from 0 to 2n - 1, so a `u8` can store numbers from 0 to 28 - 1, which equals 0 to 255.

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 is capable of more 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.

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

```
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 remainder. The following code shows how you'd use each one in a `let` statement:

```
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;

    // 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`. The boolean type in Rust is specified using `bool`. For example:

```
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.

The Character Type

So far we'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:

```
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 a variety of types into one compound type.

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:

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

The variable `tup` binds to the entire tuple, since 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:

```
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`.

In addition to destructuring through pattern matching, 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:

```
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 variables for each element by using their index. As with most programming languages, the first index in a tuple is 0.

Arrays

Another way to have a collection of multiple values is with an *array*. Unlike a tuple, every element of an array must have the same type. Arrays in Rust are different 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:

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

Arrays are useful when you want your data allocated on the stack 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. They aren't as flexible as the vector type, though. 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 in a program that needs to know the names of the months of the year. It's very unlikely that such a program will need to add or remove months, so you can use an array because you know it will always contain 12 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:

```
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

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:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

    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:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

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 9 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:

```
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.

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.

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 Parameters

Functions can also be defined to have *parameters*, which are special variables that are part of a function's signature. When a function has parameters, we 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.

The following rewritten version of `another_function` shows what parameters look like in Rust:

```
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
```

The declaration of `another_function` has one parameter named `x`. 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 of each parameter. This is a deliberate decision in Rust's design: requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what you mean.

When you want a function to have multiple parameters, separate the parameter declarations with commas, like this:

```
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 parameters, both of which are `i32` types. The function then prints out the values in both of its parameters. Note that function parameters don't all need to be the same type, they just happen to be in this example.

Let's try running this code. Replace the program currently in your *function* 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)
     Running `target/debug/functions`
The value of x is: 5
The value of y is: 6
```

Because we called the function with `5` as the value for `x` and `6` is passed as the value for `y`, 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, we've only covered 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

We've actually 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 Listing 3-1, `let y = 6;` is a statement:

```
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.

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

```
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
```

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 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 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. The block that we use to create new scopes, {}, is an expression, for example:

```
fn main() {
    let x = 5;

    ❶ let y = { ❷
        let x = 3;
        x + 1 ❸
    };

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

The expression ❷ is a block that, in this case, evaluates to 4. That value gets bound to y as part of the let statement ❶. Note the line without a semicolon at the end ❸, 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, 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. We don't name return values, but we 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:

```
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 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:

```
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?

```
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:

```
error[E0308]: mismatched types
 --> src/main.rs:7:28
  |
7 |   fn plus_one(x: i32) -> i32 {
  |  ____________________________^ starting here...
8 | |     x + 1;
9 | | }
  | |_^ ...ending here: expected i32, found ()
  |
  = note: expected type `i32`
             found type `()`
help: consider removing this semicolon:
 --> src/main.rs:8:10
```

```
  |
8 |     x + 1;
  |          ^
```

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 empty tuple. 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 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 continue 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:

```
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:

```
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:

```
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. 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 on page XX 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)
     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:

```
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:

```
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected bool, found integral variable
  |
  = note: expected type `bool`
             found type `{integer}`
```

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 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:

```
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:

```
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. That's because 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 let statement

Because `if` is an expression, we can use it on the right side of a `let` statement, like in Listing 3-2:

```
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)
     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. But what happens if the types are mismatched, as in the following example?

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

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

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

When we try to 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:

```
error[E0308]: if and else have incompatible types
 --> src/main.rs:4:18
  |
4 |       let number = if condition {
  |  __________________^ starting here...
5 | |         5
6 | |     } else {
7 | |         “six”
8 | |     };
  | |_____^ ...ending here: expected integral variable, found reference
  |
  = note: expected type `{integer}`
             found type `&’static str`
```

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` variable 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:

```
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 the "Quitting After a Correct Guess" section of Chapter 2 on page XX 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 built-in 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:

```
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:

```
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;
    }
}
```

Listing 3-3: 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 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 `5` 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 adds runtime code 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:

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

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

Listing 3-4: 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 3-3. 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.

For example, in the code in Listing 3-3, 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.

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`, 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:

```
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.

4

UNDERSTANDING OWNERSHIP

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector. Therefore, it's important to understand how ownership works in Rust. In this chapter we'll talk about ownership as well as several related features: borrowing, slices, and how Rust lays data out in memory.

What Is Ownership?

Rust's central feature is *ownership*. Although the feature is straightforward to explain, it has deep implications for the rest of the language.

All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages,

the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. No run-time costs are incurred for any of the ownership features.

Because ownership is a new concept for many programmers, it does take some time to get used to. The good news is that the more experienced you become with Rust and the rules of the ownership system, the more you'll be able to naturally develop code that is safe and efficient. Keep at it!

When you understand ownership, you'll have a solid foundation for understanding the features that make Rust unique. In this chapter, you'll learn ownership by working through some examples that focus on a very common data structure: strings.

THE STACK AND THE HEAP

In many programming languages, we don't have to think about the stack and the heap very often. But in a systems programming language like Rust, whether a value is on the stack or the heap has more of an effect on how the language behaves and why we have to make certain decisions. We'll describe parts of ownership in relation to the stack and the heap later in this chapter, so here is a brief explanation in preparation.

Both the stack and the heap are parts of memory that is available to your code to use at runtime, but they are structured in different ways. The stack stores values in the order it gets them and removes the values in the opposite order. This is referred to as *last in, first out*. Think of a stack of plates: when you add more plates, you put them on top of the pile, and when you need a plate, you take one off the top. Adding or removing plates from the middle or bottom wouldn't work as well! Adding data is called *pushing onto the stack*, and removing data is called *popping off the stack*.

The stack is fast because of the way it accesses the data: it never has to search for a place to put new data or a place to get data from because that place is always the top. Another property that makes the stack fast is that all data on the stack must take up a known, fixed size.

For data with a size unknown to us at compile time or a size that might change, we can store data on the heap instead. The heap is less organized: when we put data on the heap, we ask for some amount of space. The operating system finds an empty spot somewhere in the heap that is big enough, marks it as being in use, and returns to us a pointer to that location. This process is called *allocating on the heap*, and sometimes we abbreviate the phrase as just "allocating." Pushing values onto the stack is not considered allocating. Because the pointer is a known, fixed size, we can store the pointer on the stack, but when we want the actual data, we have to follow the pointer.

Think of being seated at a restaurant. When you enter, you state the number of people in your group, and the staff finds an empty table that fits everyone and leads you there. If someone in your group comes late, they can ask where you've been seated to find you.

Accessing data in the heap is slower than accessing data on the stack because we have to follow a pointer to get there. Contemporary processors are faster if they jump around less in memory. Continuing the analogy, consider a server at a restaurant taking orders from many tables. It's most efficient to get all the orders at one table before moving on to the next table. Taking an order from table A, then an order from table B, then one from A again, and then one from B again would be a much slower process. By the same token, a processor can do its job better if it works on data that's close to other data (as it is on the stack) rather than farther away (as it can be on the heap). Allocating a large amount of space on the heap can also take time.

When our code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function's local variables get pushed onto the stack. When the function is over, those values get popped off the stack.

Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap so we don't run out of space are all problems that ownership addresses. Once you understand ownership, you won't need to think about the stack and the heap very often, but knowing that managing heap data is why ownership exists can help explain why it works the way it does.

Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate the rules:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Variable Scope

We've walked through an example of a Rust program already in Chapter 2. Now that we're past basic syntax, we won't include all the `fn main() {` code in examples, so if you're following along, you'll have to put the following examples inside a `main` function manually. As a result, our examples will be a bit more concise, letting us focus on the actual details rather than boilerplate code.

As a first example of ownership, we'll look at the *scope* of some variables. A scope is the range within a program for which an item is valid. Let's say we have a variable that looks like this:

```
let s = "hello";
```

The variable `s` refers to a string literal, where the value of the string is hardcoded into the text of our program. The variable is valid from the point at which it's declared until the end of the current *scope*. Listing 4-1 has comments annotating where the variable `s` is valid:

```
{                      // s is not valid here, it's not yet declared
    let s = "hello";   // s is valid from this point forward

    // do stuff with s
}                      // this scope is now over, and s is no longer valid
```

Listing 4-1: A variable and the scope in which it is valid

In other words, there are two important points in time here:

1. When `s` comes *into scope*, it is valid.
2. It remains so until it goes *out of scope*.

At this point, the relationship between scopes and when variables are valid is similar to other programming languages. Now we'll build on top of this understanding by introducing the `String` type.

The String Type

To illustrate the rules of ownership, we need a data type that is more complex than the ones we covered in Chapter 3. All the data types we've looked at previously are stored on the stack and popped off the stack when their scope is over, but we want to look at data that is stored on the heap and explore how Rust knows when to clean up that data.

We'll use `String` as the example here and concentrate on the parts of `String` that relate to ownership. These aspects also apply to other complex data types provided by the standard library and that you create. We'll discuss `String` in more depth in Chapter 8.

We've already seen string literals, where a string value is hardcoded into our program. String literals are convenient, but they aren't always suitable for every situation in which you want to use text. One reason is that they're immutable. Another is that not every string value can be known when we write our code: for example, what if we want to take user input and store it? For these situations, Rust has a second string type, `String`. This type is allocated on the heap and as such is able to store an amount of text that is unknown to us at compile time. You can create a `String` from a string literal using the `from` function, like so:

```
let s = String::from("hello");
```

The double colon (`::`) is an operator that allows us to namespace this particular `from` function under the `String` type rather than using some sort of name like `string_from`. We'll discuss this syntax more in the "Method Syntax" section of Chapter 5 and when we talk about namespacing with modules in Chapter 7.

This kind of string *can* be mutated:

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

So, what's the difference here? Why can `String` be mutated but literals cannot? The difference is how these two types deal with memory.

Memory and Allocation

In the case of a string literal, we know the contents at compile time so the text is hardcoded directly into the final executable, making string literals fast and efficient. But these properties only come from its immutability. Unfortunately, we can't put a blob of memory into the binary for each piece of text whose size is unknown at compile time and whose size might change while running the program.

With the `String` type, in order to support a mutable, growable piece of text, we need to allocate an amount of memory on the heap, unknown at compile time, to hold the contents. This means:

1. The memory must be requested from the operating system at runtime.
2. We need a way of returning this memory to the operating system when we're done with our `String`.

That first part is done by us: when we call `String::from`, its implementation requests the memory it needs. This is pretty much universal in programming languages.

However, the second part is different. In languages with a *garbage collector (GC)*, the GC keeps track and cleans up memory that isn't being used anymore, and we, as the programmer, don't need to think about it. Without a GC, it's the programmer's responsibility to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it. Doing this correctly has historically been a difficult programming problem. If we forget, we'll waste memory. If we do it too early, we'll have an invalid variable. If we do it twice, that's a bug too. We need to pair exactly one `allocate` with exactly one `free`.

Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope. Here's a version of our scope example from Listing 4-1 using a `String` instead of a string literal:

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid
```

There is a natural point at which we can return the memory our `String` needs to the operating system: when `s` goes out of scope. When a variable goes out of scope, Rust calls a special function for us. This function is called `drop`, and it's where the author of `String` can put the code to return the memory. Rust calls `drop` automatically at the closing }.

NOTE *In C++, this pattern of deallocating resources at the end of an item's lifetime is sometimes called* Resource Acquisition Is Initialization (RAII). *The `drop` function in Rust will be familiar to you if you've used RAII patterns.*

This pattern has a profound impact on the way Rust code is written. It may seem simple right now, but the behavior of code can be unexpected in more complicated situations when we want to have multiple variables use the data we've allocated on the heap. Let's explore some of those situations now.

Ways Variables and Data Interact: Move

Multiple variables can interact with the same data in different ways in Rust. Let's look at an example using an integer in Listing 4-2:

```
let x = 5;
let y = x;
```

Listing 4-2: Assigning the integer value of variable `x` to `y`

We can probably guess what this is doing based on our experience with other languages: "Bind the value `5` to `x`; then make a copy of the value in `x` and bind it to `y`." We now have two variables, `x` and `y`, and both equal `5`. This is indeed what is happening because integers are simple values with a known, fixed size, and these two `5` values are pushed onto the stack.

Now let's look at the `String` version:

```
let s1 = String::from("hello");
let s2 = s1;
```

This looks very similar to the previous code, so we might assume that the way it works would be the same: that is, the second line would make a copy of the value in `s1` and bind it to `s2`. But this isn't quite what happens.

To explain this more thoroughly, let's look at what `String` looks like under the covers in Figure 4-1. A `String` is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

The length is how much memory, in bytes, the contents of the `String` is currently using. The capacity is the total amount of memory, in bytes, that the `String` has received from the operating system. The difference between length and capacity matters, but not in this context, so for now, it's fine to ignore the capacity.

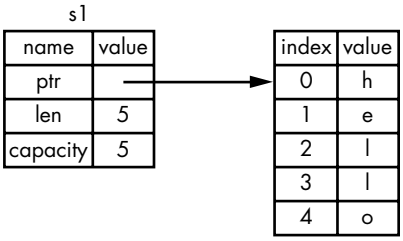

Figure 4-1: Representation in memory of a `String` holding the value `"hello"` bound to `s1`

When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to. In other words, the data representation in memory looks like Figure 4-2.

The representation does *not* look like Figure 4-3, which is what memory would look like if Rust instead copied the heap data as well. If Rust did this, the operation `s2 = s1` could potentially be very expensive in terms of runtime performance if the data on the heap was large.

Earlier, we said that when a variable goes out of scope, Rust automatically calls the `drop` function and cleans up the heap memory for that variable. But Figure 4-2 shows both data pointers pointing to the same location. This is

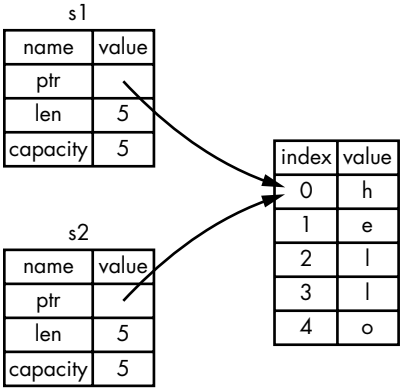

Figure 4-2: Representation in memory of the variable `s2` that has a copy of the pointer, length, and capacity of `s1`

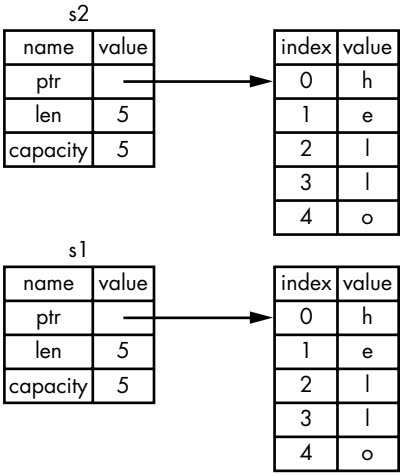

Figure 4-3: Another possibility of what `s2 = s1` might do if Rust copied the heap data as well

a problem: when `s2` and `s1` go out of scope, they will both try to free the same memory. This is known as a *double free* error and is one of the memory safety bugs we mentioned previously. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

To ensure memory safety, there's one more detail to what happens in this situation in Rust. Instead of trying to copy the allocated memory, Rust considers `s1` to no longer be valid and therefore, Rust doesn't need to free anything when `s1` goes out of scope. Check out what happens when you try to use `s1` after `s2` is created:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}", s1);
```

You'll get an error like this because Rust prevents you from using the invalidated reference:

```
5:22 error: use of moved value: `s1` [E0382]
println!("{}", s1);
                ^~
5:24 note: in this expansion of println! (defined in <std macros>)
3:11 note: `s1` moved here because it has type `collections::string::String`,
which is moved by default
 let s2 = s1;
     ^~
```

If you've heard the terms "shallow copy" and "deep copy" while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like a shallow copy. But because Rust also invalidates the first variable, instead of calling this a shallow copy, it's known as a *move*. Here we would read this by saying that `s1` was *moved* into `s2`. So what actually happens is shown in Figure 4-4.

That solves our problem! With only `s2` valid, when it goes out of scope, it alone will free the memory, and we're done.

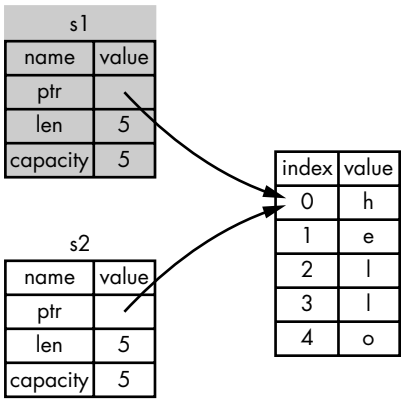

Figure 4-4: Representation in memory after `s1` has been invalidated

In addition, there's a design choice that's implied by this: Rust will never automatically create "deep" copies of your data. Therefore, any *automatic* copying can be assumed to be inexpensive in terms of runtime performance.

Ways Variables and Data Interact: Clone

If we *do* want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called `clone`. We'll discuss method syntax in Chapter 5, but because methods are a common feature in many programming languages, you've probably seen them before.

Here's an example of the `clone` method in action:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

This works just fine and is how you can explicitly produce the behavior shown in Figure 4-3, where the heap data *does* get copied.

When you see a call to `clone`, you know that some arbitrary code is being executed and that code may be expensive. It's a visual indicator that something different is going on.

Stack-Only Data: Copy

There's another wrinkle we haven't talked about yet. This code using integers, part of which was shown earlier in Listing 4-2, works and is valid:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

But this code seems to contradict what we just learned: we don't have a call to `clone`, but `x` is still valid and wasn't moved into `y`.

The reason is that types like integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make. That means there's no reason we would want to prevent `x` from being valid after we create the variable `y`. In other words, there's no difference between deep and shallow copying here, so calling `clone` wouldn't do anything differently from the usual shallow copying and we can leave it out.

Rust has a special annotation called the `Copy` trait that we can place on types like integers that are stored on the stack (we'll talk more about traits in Chapter 10). If a type has the `Copy` trait, an older variable is still usable after assignment. Rust won't let us annotate a type with the `Copy` trait if the type, or any of its parts, has implemented the `Drop` trait. If the type needs something special to happen when the value goes out of scope and we add the `Copy` annotation to that type, we'll get a compile time error.

So what types are `Copy`? You can check the documentation for the given type to be sure, but as a general rule, any group of simple scalar values can be `Copy`, and nothing that requires allocation or is some form of resource is `Copy`. Here are some of the types that are `Copy`:

- All the integer types, like `u32`.
- The boolean type, `bool`, with values `true` and `false`.
- All the floating point types, like `f64`.
- Tuples, but only if they contain types that are also `Copy`. `(i32, i32)` is `Copy`, but `(i32, String)` is not.

Ownership and Functions

The semantics for passing a value to a function are similar to assigning a value to a variable. Passing a variable to a function will move or copy, just like assignment. Listing 4-3 has an example with some annotations showing where variables go into and out of scope:

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello");  // s comes into scope.

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here.
    let x = 5;                      // x comes into scope.

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward.

} // Here, x goes out of scope, then s. But since s's value was moved,
  // nothing
 special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope.
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope.
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Listing 4-3: Functions with ownership and scope annotated

If we tried to use `s` after the call to `takes_ownership`, Rust would throw a compile time error. These static checks protect us from mistakes. Try adding code to `main` that uses `s` and `x` to see where you can use them and where the ownership rules prevent you from doing so.

Return Values and Scope

Returning values can also transfer ownership. Here's an example with similar annotations to those in Listing 4-3:

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1.

    let s2 = String::from("hello");     // s2 comes into scope.

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3.
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was
  // moved, so nothing happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it.

    let some_string = String::from("hello"); // some_string comes into scope.

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function.
}

// takes_and_gives_back will take a String and return one.
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
scope.

    a_string  // a_string is returned and moves out to the calling function.
}
```

The ownership of variables follows the same pattern every time: assigning a value to another variable moves it, and when heap data values' variables go out of scope, if the data hasn't been moved to be owned by another variable, the value will be cleaned up by `drop`.

Taking ownership and then returning ownership with every function is a bit tedious. What if we want to let a function use a value but not take ownership? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.

It's possible to return multiple values using a tuple, like this:

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}
```

```
fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String.

    (s, length)
}
```

But this is too much ceremony and a lot of work for a concept that should be common. Luckily for us, Rust has a feature for this concept, and it's called *references*.

References and Borrowing

The issue with the tuple code at the end of the preceding section is that we have to return the `String` to the calling function so we can still use the `String` after the call to `calculate_length`, because the `String` was moved into `calculate_length`.

Here is how you would define and use a `calculate_length` function that takes a *reference* to an object as an argument instead of taking ownership of the argument:

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

First, notice that all the tuple code in the variable declaration and the function return value is gone. Second, note that we pass `&s1` into `calculate_length`, and in its definition, we take `&String` rather than `String`.

These ampersands are *references*, and they allow you to refer to some value without taking ownership of it. Figure 4-5 shows a diagram.

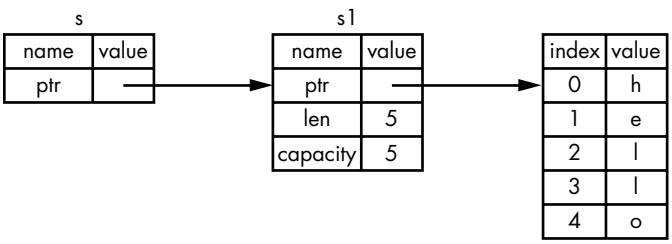

Figure 4-5: `&String s` pointing at `String s1`

Let's take a closer look at the function call here:

```
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

The `&s1` syntax lets us create a reference that *refers* to the value of `s1` but does not own it. Because it does not own it, the value it points to will not be dropped when the reference goes out of scope.

Likewise, the signature of the function uses `&` to indicate that it takes a reference as an argument. Let's add some explanatory annotations:

```
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of
  // what
 it refers to, nothing happens.
```

The scope in which the variable `s` is valid is the same as any function argument's scope, but we don't drop what the reference points to when it goes out of scope because we don't have ownership. Functions that take references as arguments instead of the actual values mean we won't need to return the values in order to give back ownership, since we never had ownership.

We call taking references as function arguments *borrowing*. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back.

So what happens if we try to modify something we're borrowing? Try the code in Listing 4-4. Spoiler alert: it doesn't work!

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Listing 4-4: Attempting to modify a borrowed value

Here's the error:

```
error: cannot borrow immutable borrowed content `*some_string` as mutable
 --> error.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^
```

Just as variables are immutable by default, so are references. We're not allowed to modify something we have a reference to.

Mutable References

We can fix the error in the code from Listing 4-4 with just a small tweak:

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

First, we had to change `s` to be `mut`. Then we had to create a mutable reference with `&mut s` and accept a mutable reference with `some_string: &mut String`.

But mutable references have one big restriction: you can only have one mutable reference to a particular piece of data in a particular scope. This code will fail:

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

Here's the error:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> borrow_twice.rs:5:19
  |
4 |     let r1 = &mut s;
  |                   - first mutable borrow occurs here
5 |     let r2 = &mut s;
  |                   ^ second mutable borrow occurs here
6 | }
  | - first borrow ends here
```

This restriction allows for mutation but in a very controlled fashion. It's something that new Rustaceans struggle with, because most languages let you mutate whenever you'd like. The benefit of having this restriction is that Rust can prevent data races at compile time.

A *data race* is a particular type of race condition in which these three behaviors occur:

1. Two or more pointers access the same data at the same time.
2. At least one of the pointers is being used to write to the data.
3. There's no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime; Rust prevents this problem from happening because it won't even compile code with data races!

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not *simultaneous* ones:

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with no
  // problems.

let r2 = &mut s;
```

A similar rule exists for combining mutable and immutable references. This code results in an error:

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM
```

Here's the error:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as
immutable
 --> borrow_thrice.rs:6:19
  |
4 |     let r1 = &s; // no problem
  |               - immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |                   ^ mutable borrow occurs here
7 | }
  | - immutable borrow ends here
```

Whew! We *also* cannot have a mutable reference while we have an immutable one. Users of an immutable reference don't expect the values to suddenly change out from under them! However, multiple immutable references are okay because no one who is just reading the data has the ability to affect anyone else's reading of the data.

Even though these errors may be frustrating at times, remember that it's the Rust compiler pointing out a potential bug early (at compile time rather than at runtime) and showing you exactly where the problem is instead of you having to track down why sometimes your data isn't what you thought it should be.

Dangling References

In languages with pointers, it's easy to erroneously create a *dangling pointer*, a pointer that references a location in memory that may have been given to someone else, by freeing some memory while preserving a pointer to that memory. In Rust, by contrast, the compiler guarantees that references will never be dangling references: if we have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

Let's try to create a dangling reference:

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Here's the error:

```
error[E0106]: missing lifetime specifier
 --> dangle.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^^^^^^^
  |
  = help: this function's return type contains a borrowed value, but there is
no
    value for it to be borrowed from
  = help: consider giving it a 'static lifetime

error: aborting due to previous error
```

This error message refers to a feature we haven't covered yet: *lifetimes*. We'll discuss lifetimes in detail in Chapter 10. But, if you disregard the parts about lifetimes, the message does contain the key to why this code is a problem:

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from.
```

Let's take a closer look at exactly what's happening at each stage of our `dangle` code:

```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
```

```
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

Because `s` is created inside `dangle`, when the code of `dangle` is finished, `s` will be deallocated. But we tried to return a reference to it. That means this reference would be pointing to an invalid `String`! That's no good. Rust won't let us do this.

The correct code here is to return the `String` directly:

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

This works without any problems. Ownership is moved out, and nothing is deallocated.

The Rules of References

Let's recap what we've discussed about references:

1. At any given time, you can have *either* but not both of:
   - One mutable reference.
   - Any number of immutable references.
2. References must always be valid.

Next, we'll look at a different kind of reference: slices.

Slices

Another data type that does not have ownership is the *slice*. Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection.

Here's a small programming problem: write a function that takes a string and returns the first word it finds in that string. If the function doesn't find a space in the string, it means the whole string is one word, so the entire string should be returned.

Let's think about the signature of this function:

```
fn first_word(s: &String) -> ?
```

This function, `first_word`, takes a `&String` as an argument. We don't want ownership, so this is fine. But what should we return? We don't really have a way to talk about *part* of a string. However, we could return the index of the end of the word. Let's try that as shown in Listing 4-5:

```
fn first_word(s: &String) -> usize {
    ❶ let bytes = s.as_bytes();
```

```
    for ❷ (i, &item) in ❸ bytes.iter().enumerate() {
        ❹ if item == b' ' {
            return i;
        }
    }

    ❺ s.len()
}
```

Listing 4-5: The `first_word` *function that returns a byte index value into the* `String` *argument*

Because we need to go through the `String` element by element and check whether a value is a space, we'll convert our `String` to an array of bytes using the `as_bytes` method ❶. Next, we create an iterator over the array of bytes using the `iter` method ❸.

We'll discuss iterators in more detail in Chapter 16. For now, know that `iter` is a method that returns each element in a collection, and `enumerate` wraps the result of `iter` and returns each element as part of a tuple instead. The first element of the tuple returned from `enumerate` is the index, and the second element is a reference to the element. This is a bit more convenient than calculating the index ourselves.

Because the `enumerate` method returns a tuple, we can use patterns to destructure that tuple, just like everywhere else in Rust. So in the `for` loop, we specify a pattern that has `i` for the index in the tuple and `&item` for the single byte in the tuple ❷. Because we get a reference to the element from `.iter().enumerate()`, we use `&` in the pattern.

Inside the `for` loop, we search for the byte that represents the space by using the byte literal syntax ❹. If we find a space, we return the position. Otherwise, we return the length of the string by using `s.len()` ❺.

We now have a way to find out the index of the end of the first word in the string, but there's a problem. We're returning a `usize` on its own, but it's only a meaningful number in the context of the `&String`. In other words, because it's a separate value from the `String`, there's no guarantee that it will still be valid in the future. Consider the program in Listing 4-6 that uses the `first_word` function from Listing 4-5:

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5.

    s.clear(); // This empties the String, making it equal to "".
```

```
    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally
    // invalid!
}
```

Listing 4-6: Storing the result from calling the `first_word` *function then changing the* `String` *contents*

This program compiles without any errors and also would if we used `word` after calling `s.clear()`. `word` isn't connected to the state of `s` at all, so `word` still contains the value `5`. We could use that value `5` with the variable `s` to try to extract the first word out, but this would be a bug because the contents of `s` have changed since we saved `5` in `word`.

Having to worry about the index in `word` getting out of sync with the data in `s` is tedious and error prone! Managing these indices is even more brittle if we write a `second_word` function. Its signature would have to look like this:

```
fn second_word(s: &String) -> (usize, usize) {
```

Now we're tracking a start *and* an ending index, and we have even more values that were calculated from data in a particular state but aren't tied to that state at all. We now have three unrelated variables floating around that need to be kept in sync.

Luckily, Rust has a solution to this problem: string slices.

String Slices

A *string slice* is a reference to part of a `String`, and looks like this:

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

This is similar to taking a reference to the whole `String` but with the extra `[0..5]` bit. Rather than a reference to the entire `String`, it's a reference to an internal position in the `String` and the number of elements that it refers to.

We create slices with a range of `[starting_index..ending_index]`, but the slice data structure actually stores the starting position and the length of the slice. So in the case of `let world = &s[6..11];`, `world` would be a slice that contains a pointer to the 6th byte of `s` and a length value of 5.

Figure 4-6 shows this in a diagram.

With Rust's `..` range syntax, if you want to start at the first index (zero), you can drop the value before the two periods. In other words, these are equal:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

By the same token, if your slice includes the last byte of the `String`, you can drop the trailing number. That means these are equal:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

You can also drop both values to take a slice of the entire string. So these are equal:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

With all this information in mind, let's rewrite `first_word` to return a slice. The type that signifies "string slice" is written as `&str`:

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

We get the index for the end of the word in the same way as we did in Listing 4-5, by looking for the first occurrence of a space. When we find a space, we return a string slice using the start of the string and the index of the space as the starting and ending indices.

Now when we call `first_word`, we get back a single value that is tied to the underlying data. The value is made up of a reference to the starting point of the slice and the number of elements in the slice.

Returning a slice would also work for a `second_word` function:

```
fn second_word(s: &String) -> &str {
```

We now have a straightforward API that's much harder to mess up, since the compiler will ensure the references into the `String` remain valid. Remember the bug in the program in Listing 4-6, when we got the index to the end of the first word but then cleared the string so our index was invalid? That code was logically incorrect but didn't show any immediate errors. The problems would show up later if we kept trying to use the first word index with an emptied string. Slices make this bug impossible and let us know we have a problem with our code much sooner. Using the slice version of `first_word` will throw a compile time error:

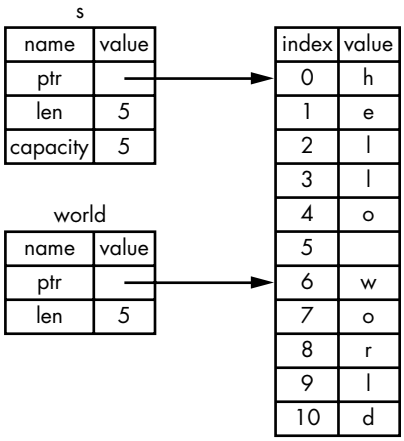

Figure 4-6: Representation in memory after s1 has been invalidated

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // Error!
}
```

Here's the compiler error:

```
17:6 error: cannot borrow `s` as mutable because it is also borrowed as
            immutable [E0502]
    s.clear(); // Error!
    ^
15:29 note: previous borrow of `s` occurs here; the immutable borrow prevents
            subsequent moves or mutable borrows of `s` until the borrow ends
    let word = first_word(&s);
                          ^
18:2 note: previous borrow ends here
fn main() {

}
^
```

Recall from the borrowing rules that if we have an immutable reference to something, we cannot also take a mutable reference. Because `clear` needs to truncate the `String`, it tries to take a mutable reference, which fails. Not only has Rust made our API easier to use, but it has also eliminated an entire class of errors at compile time!

String Literals Are Slices

Recall that we talked about string literals being stored inside the binary. Now that we know about slices, we can properly understand string literals:

```
let s = "Hello, world!";
```

The type of `s` here is `&str`: it's a slice pointing to that specific point of the binary. This is also why string literals are immutable; `&str` is an immutable reference.

String Slices as Arguments

Knowing that you can take slices of literals and `String`s leads us to one more improvement on `first_word`, and that's its signature:

```
fn first_word(s: &String) -> &str {
```

A more experienced Rustacean would write the following line instead because it allows us to use the same function on both `String`s and `&str`s:

```
fn first_word(s: &str) -> &str {
```

If we have a string slice, we can pass that as the argument directly. If we have a `String`, we can pass a slice of the entire `String`. Defining a function to take a string slice argument instead of a reference to a String makes our API more general and useful without losing any functionality:

```
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // since string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Other Slices

String slices, as you might imagine, are specific to strings. But there's a more general slice type, too. Consider this array:

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

Just like we might want to refer to a part of a string, we might want to refer to part of an array and would do so like this:

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

let slice = &a[1..3];
```

This slice has the type `&[i32]`. It works the same way as string slices do, by storing a reference to the first element and a length. You'll use this kind of slice for all sorts of other collections. We'll discuss these collections in detail when we talk about vectors in Chapter 8.

Summary

The concepts of ownership, borrowing, and slices are what ensure memory safety in Rust programs at compile time. The Rust language gives you control over your memory usage like other systems programming languages, but having the owner of data automatically clean up that data when the owner goes out of scope means you don't have to write and debug extra code to get this control.

Ownership affects how lots of other parts of Rust work, so we'll talk about these concepts further throughout the rest of the book. Let's move on to the next chapter and look at grouping pieces of data together in a `struct`.