Programming a 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, external crates, and more! In the following chapters, we’ll explore these ideas in more detail. In this chapter, you’ll just 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 a guess is entered, the program will indicate whether the guess is too low or too high. If the guess is correct, the game will print a congratulatory message 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

$ cd guessing\_game

The first command, cargo new, takes the name of the project (guessing\_game) as the first argument. The second command changes to the new project’s directory.

Look at the generated Cargo.toml file:

Cargo.toml

[package]

name = "guessing\_game"

version = "0.1.0"

edition = "2024"

[dependencies]

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

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)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.50s

Running `target/debug/guessing\_game`

Hello, world!

The run command comes in handy when you need to rapidly iterate on a project, as we’ll do in this game, quickly testing 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 guessing game 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.

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}");

}

Code that gets a guess from the user and prints it

This code contains a lot of information, so let’s go over it line by line. To obtain user input and then print the result as output, we need to bring the io input/output library into scope. The io library comes from the standard library, known as std:

use std::io;

By default, Rust has a set of items defined in the standard library that it brings into the scope of every program. This set is called the prelude, and you can see everything in it in the standard library documentation at https://doc.rust-lang.org/std/prelude/index.html.

If a type you want to use isn’t in the prelude, you have to bring that type into scope explicitly with a use statement. Using the std::io library provides you with a number of useful features, including the ability 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 parentheses, (), indicate there are no parameters; and the curly bracket, {, 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 printing a prompt stating what the game is and requesting input from the user.

Storing Values with Variables

Next, we’ll create a variable 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. We use the let statement to create the variable. Here’s another example:

let apples = 5;

This line creates a new variable named apples and binds it to the value 5. In Rust, variables are immutable by default, meaning once we give the variable a value, the value won’t change. We’ll be discussing this concept in detail in “Variables and Mutability” on page XX. To make a variable mutable, we add mut before the variable name:

let apples = 5; // immutable

let mut bananas = 5; // mutable

Note The // syntax starts a comment that continues until the end of the line. Rust ignores everything in comments. We’ll discuss comments in more detail in Chapter 3.

Returning to the guessing game program, you now know that let mut guess will introduce a mutable variable named guess. The equal sign (=) tells Rust we want to bind something to the variable now. On the right 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 a function that’s implemented on a type, in this case String. 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.

In full, 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!

Receiving User Input

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 the stdin function from the io module, which will allow us to handle user input:

io::stdin()

.read\_line(&mut guess)

If we hadn’t imported the io module with use std::io; at the beginning of the program, we could still use the function by writing 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.

Next, the line .read\_line(&mut guess) calls the read\_line method on the standard input handle to get input from the user. We’re also passing &mut guess as the argument to read\_line to tell it what string to store the user input in. The full job of read\_line is to take whatever the user types into standard input and append that into a string (without overwriting its contents), so we therefore pass that string as an argument. The string argument needs to be mutable so that the method can change the string’s content.

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. 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. (Chapter 4 will explain references more thoroughly.)

Handling Potential Failure with Result

We’re still working on this line of code. We’re now discussing a third line of text, but note that it’s still part of a single logical line of code. The next part is this method:

.expect("Failed to read line");

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. It’s often wise to introduce a newline and other whitespace to help break up long lines when you call a method with the .method\_name() syntax. Now let’s discuss what this line does.

As mentioned earlier, read\_line puts whatever the user enters into the string we pass to it, but it also returns a Result value. Result is an enumeration, often called an enum, which is a type that can be in one of multiple possible states. We call each possible state a variant.

Chapter 6 will cover enums in more detail. The purpose of these Result types is to encode error-handling information.

Result’s variants are Ok and Err. The Ok variant indicates the operation was successful, and it contains the successfully generated value. The Err variant means the operation failed, and it contains information about how or why the operation failed.

Values of the Result type, like values of any type, have methods defined on them. An instance of Result has an expect method that you can call. If this instance of Result is an Err value, expect will cause the program to crash and display the message that you passed as an argument to 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 Result is an Ok value, expect will take the return value that Ok is holding and return just that value to you so that you can use it. In this case, that value is the number of bytes in the user’s input.

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

$ cargo build

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

warning: unused `Result` that must be used

--> src/main.rs:10:5

|

10 | io::stdin().read\_line(&mut guess);

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

|

= note: this `Result` may be an `Err` variant, which should be handled

= note: `#[warn(unused\_must\_use)]` on by default

help: use `let \_ = ...` to ignore the resulting value

|

10 | let \_ = io::stdin().read\_line(&mut guess);

| +++++++

warning: `guessing\_game` (bin "guessing\_game") generated 1 warning

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.59s

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 code, but in our case we just want to crash this program when a problem occurs, so we can use expect. You’ll learn about recovering from errors in Chapter 9.

Printing Values with println! Placeholders

Aside from the closing curly bracket, there’s only one more line to discuss in the code so far:

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

This line prints the string that now contains the user’s input. The {} set of curly brackets is a placeholder: Think of {} as little crab pincers that hold a value in place. When printing the value of a variable, the variable name can go inside the curly brackets. When printing the result of evaluating an expression, place empty curly brackets in the format string, then follow the format string with a comma-separated list of expressions to print in each empty curly bracket placeholder in the same order. Printing a variable and the result of an expression in one call to println! would look like this:

let x = 5;

let y = 10;

println!("x = {x} and y + 2 = {}", y + 2);

This code would print x = 5 and y + 2 = 12.

Testing the First Part

Let’s test the first part of the guessing game. Run it using cargo run:

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 6.44s

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 that the game is fun to play more than once. We’ll use a random number between 1 and 100 so that 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 with said functionality.

Increasing Functionality with a Crate

Remember that a crate is a collection of Rust source code files. The project we’ve been building is a binary crate, which is an executable. The rand crate is a library crate, which contains code that is intended to be used in other programs and can’t be executed on its own.

Cargo’s coordination of external crates is where Cargo 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. Be sure to specify rand exactly as we have here, with this version number, or the code examples in this tutorial may not work:

Cargo.toml

[dependencies]

rand = "0.8.5"

In the Cargo.toml file, everything that follows a header is part of that section that continues until another section starts. In [dependencies], you tell Cargo which external crates your project depends on and which versions of those crates you require. In this case, we specify the rand crate with the semantic version specifier 0.8.5. Cargo understands Semantic Versioning (sometimes called SemVer), which is a standard for writing version numbers. The specifier 0.8.5 is actually shorthand for ^0.8.5, which means any version that is at least 0.8.5 but below 0.9.0.

Cargo considers these versions to have public APIs compatible with version 0.8.5, and this specification ensures that you’ll get the latest patch release that will still compile with the code in this chapter. Any version 0.9.0 or greater is not guaranteed to have the same API as what the following examples use.

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

$ cargo build

Updating crates.io index

Locking 15 packages to latest Rust 1.85.0 compatible versions

Adding rand v0.8.5 (available: v0.9.0)

Compiling proc-macro2 v1.0.93

Compiling unicode-ident v1.0.17

Compiling libc v0.2.170

Compiling cfg-if v1.0.0

Compiling byteorder v1.5.0

Compiling getrandom v0.2.15

Compiling rand\_core v0.6.4

Compiling quote v1.0.38

Compiling syn v2.0.98

Compiling zerocopy-derive v0.7.35

Compiling zerocopy v0.7.35

Compiling ppv-lite86 v0.2.20

Compiling rand\_chacha v0.3.1

Compiling rand v0.8.5

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.48s

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 different lines (depending on the operating system), and the lines may be in a different order.

When we include an external dependency, Cargo fetches the latest versions of everything that dependency needs from the registry, which is a copy of data from Crates.io at 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 crates listed that aren’t already downloaded. In this case, although we only listed rand as a dependency, Cargo also grabbed other crates that rand depends on to work. After downloading the crates, 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 aside from the Finished line. 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 the src/main.rs file, make a trivial change, and then save it and build again, you’ll only see two lines of output:

$ cargo build

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.53 secs

These lines show that 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.

Ensuring Reproducible Builds

Cargo has a mechanism that ensures that 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, say that next week version 0.8.6 of the rand crate comes out, and that version contains an important bug fix, but it also contains a regression that will break your code. To handle this, Rust creates the Cargo.lock file the first time you run cargo build, so we now have this in the 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 will use the versions specified there rather than doing all the work of figuring out versions again. This lets you have a reproducible build automatically. In other words, your project will remain at 0.8.5 until you explicitly upgrade, thanks to the Cargo.lock file. Because the Cargo.lock file is important for reproducible builds, it’s often checked into source control with the rest of the code in your project.

Updating a Crate to Get a New Version

When you do want to update a crate, Cargo provides the command update, which will ignore the Cargo.lock file and figure out all the latest versions that fit your specifications in Cargo.toml. Cargo will then write those versions to the Cargo.lock file. Otherwise, by default, Cargo will only look for versions greater than 0.8.5 and less than 0.9.0. If the rand crate has released the two new versions 0.8.6 and 0.9.0, you would see the following if you ran cargo update:

$ cargo update

Updating crates.io index

Locking 1 package to latest Rust 1.85.0 compatible version

Updating rand v0.8.5 -> v0.8.6 (available: v0.9.0)

Cargo ignores the 0.9.0 release. 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 0.8.6. To use rand version 0.9.0 or any version in the 0.9.x series, you’d have to update the Cargo.toml file to look like this instead:

[dependencies]

rand = "0.9.0"

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 have specified.

There’s a lot more to say about Cargo and its ecosystem, which we’ll discuss in Chapter 14, 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 to generate a number to guess. The next step is to update src/main.rs, as shown in Listing 2-3.

src/main.rs

use std::io;

1 use rand::Rng;

fn main() {

println!("Guess the number!");

2 let secret\_number = rand::thread\_rng().gen\_range(1..=100);

3 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}");

}

Adding code to generate a random number

First, we add the line use rand::Rng; 1. The Rng trait 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.

Next, we’re adding two lines in the middle. In the first line, we call the rand::thread\_rng function 2 that gives us the particular random number generator we’re going to use: one that is local to the current thread of execution and is seeded by the operating system. Then, 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 a range expression as an argument and generates a random number in the range. The kind of range expression we’re using here takes the form start..=end and is inclusive on the lower and upper bounds, so we need to specify 1..=100 to request a number between 1 and 100.

Note You won’t just know which traits to use and which methods and functions to call from a crate, so each crate has documentation with instructions for using it. Another neat feature of Cargo is that running the cargo doc --open command will build documentation provided by all your dependencies locally and open it in your browser. 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 new line prints the secret number 3. 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)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.53s

Running `target/debug/guessing\_game`

Guess the number!

The secret number is: 7

Please input your guess.

4

You guessed: 4

$ cargo run

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s

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. Note that this code won’t compile just yet, as we will explain.

src/main.rs

1 use std::cmp::Ordering;

use std::io;

use rand::Rng;

fn main() {

--snip--

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

2 match guess.cmp(&secret\_number) { 3

Ordering::Less => println!("Too small!"),

Ordering::Greater => println!("Too big!"),

Ordering::Equal => println!("You win!"),

}

}

Handling the possible return values of comparing two numbers

First, we add another use statement 1, bringing a type called std::cmp::Ordering into scope from the standard library. The Ordering type is another enum and has the variants 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 3 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 guess to secret\_number. Then, it returns a variant of the Ordering enum we brought into scope with the use statement. We use a match expression 2 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 to match against, and the code that should be run if the value given to match fits that arm’s pattern. Rust takes the value given to match and looks through each arm’s pattern in turn. Patterns and the match construct are powerful Rust features: They let you express a variety of situations your code might encounter, and they make sure you handle them all. These features will be covered in detail in Chapter 6 and Chapter 19, respectively.

Let’s walk through an example with the match expression we use 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. The match expression gets the Ordering::Greater value and starts checking each arm’s pattern. 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 is Ordering::Greater, which does match Ordering::Greater! The associated code in that arm will execute and print Too big! to the screen. The match expression ends after the first successful match, so it won’t look at the last arm in this 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:22:21

|

22 | match guess.cmp(&secret\_number) {

| --- ^^^^^^^^^^^^^^ expected `&String`, found `&{integer}`

| |

| arguments to this method are incorrect

|

= note: expected reference `&String`

found reference `&{integer}`

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 mut 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 of Rust’s 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. Unless otherwise specified, 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 cannot compare a string and a number type.

Ultimately, we want to convert the String the program reads as input into a number type so that we can compare it numerically to the secret number. We do so by adding this line to the main function body:

src/main.rs

--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 helpfully Rust allows us to shadow the previous value of guess with a new one. Shadowing lets us reuse the guess variable name rather than forcing us to create two unique variables, such as guess\_str and guess, for example. We’ll cover this in more detail in Chapter 3, but for now, know that this feature is often used when you want to convert a value from one type to another type.

We bind this new variable to the expression guess.trim().parse(). The guess in the expression refers to the original guess variable that contained the input as a string. The trim method on a String instance will eliminate any whitespace at the beginning and end, which we must do before we can convert the string to a u32, which can only contain numerical data. The user must press enter to satisfy read\_line and input their guess, which adds a newline character to the string. For example, if the user types 5 and presses enter, guess looks like this: 5\n. The \n represents “newline.” (On Windows, pressing enter results in a carriage return and a newline, \r\n.) The trim method eliminates \n or \r\n, resulting in just 5.

The parse method on strings converts a string to another type. Here, we use it to convert from a string to a number. We need to tell Rust the exact number type we want by using let guess: u32. The 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 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 parse method will only work on characters that can logically be converted into numbers and so can easily cause errors. 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 as the read\_line method does (discussed earlier in “Handling Potential Failure with Result” 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)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s

Running `target/debug/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 creates an infinite loop. We’ll add a loop to give users more chances at guessing the number:

src/main.rs

--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 from the guess input prompt onward into a loop. Be sure to indent the lines inside the loop another four spaces each and run the program again. The program will now ask for another guess forever, which actually introduces a new problem. It doesn’t seem like the user can quit!

The user could always interrupt 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 Guess to the Secret Number” on page XX: If the user enters a non-number answer, the program will crash. We can take advantage of that to allow the user to quit, as shown here:

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 1.50s

Running `target/debug/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 src/main.rs:28:47:

Please type a number!: ParseIntError { kind: InvalidDigit }

note: run with `RUST\_BACKTRACE=1` environment variable to display a backtrace

Typing quit will quit the game, but as you’ll notice, so will entering any other non-number input. This is suboptimal, to say the least; we want the game to also 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 statement:

src/main.rs

--snip--

match guess.cmp(&secret\_number) {

Ordering::Less => println!("Too small!"),

Ordering::Greater => println!("Too big!"),

Ordering::Equal => {

println!("You win!");

break;

}

}

Adding the break line after You win! makes the program 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 that the user can continue guessing. We can do that by altering the line where guess is converted from a String to a u32, as shown in Listing 2-5.

src/main.rs

--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--

Ignoring a non-number guess and asking for another guess instead of crashing the program

We switch from an expect call to a match expression to move from crashing on an error to handling the error. Remember that parse returns a Result type and Result is an enum that has the variants Ok and Err. We’re using a match expression here, as 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 resultant 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 underscore, \_, is a catch-all 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 tells the program 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:

$ cargo run

Compiling guessing\_game v0.1.0 (file:///projects/guessing\_game)

Finished `dev` profile [unoptimized + debuginfo] target(s) in 4.45s

Running `target/debug/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 the secret number. That worked well for testing, but it ruins the game. Let’s delete the println! that outputs the secret number. Listing 2-6 shows the final code.

src/main.rs

use std::cmp::Ordering;

use std::io;

use rand::Rng;

fn main() {

println!("Guess the number!");

let secret\_number = rand::thread\_rng().gen\_range(1..=100);

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;

}

}

}

}

Complete guessing game code

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

Summary

This project was a hands-on way to introduce you to many new Rust concepts: let, match, functions, the use of 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, a feature that makes Rust different from other languages. Chapter 5 discusses structs and method syntax, and Chapter 6 explains how enums work.