

2

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, 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 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"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[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:

```
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 [unoptimized + debuginfo] target(s) in 1.50 secs
    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*.

```

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 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 (which is known as `std`):

```

use std::io;

```

By default, Rust brings only a few types into the scope of every program in the *prelude*. 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 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 a *variable*. Here's another example:

```
let foo = bar;
```

This line creates a new variable named `foo` and binds it to the value of the `bar` variable. In Rust, variables are immutable by default. We'll discuss this concept in detail in "Variables and Mutability" on page 32. 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, which are discussed in more detail in Chapter 3.

Let's return to the guessing game program. You now 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 the `stdin` function from the `io` module:

```
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. 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 the Result Type

We're not quite done with this line of code. Although what we've discussed so far is 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.

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`. The `Ok` variant indicates the operation was successful, and inside `Ok` is the successfully generated value. The `Err` variant 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 values of 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 bytes in what 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)
warning: unused `std::result::Result` which must be used
--> src/main.rs:10:5
10 |         io::stdin().read_line(&mut guess);
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    = note: #[warn(unused_must_use)] on by default
```

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 because 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 the string we saved the user's input in. The set of curly brackets, `{}`, is a placeholder: think of `{}` as little crab pincers that hold a value in place. You can print more than one value using curly brackets: the first set of curly brackets holds the first value listed after the format string, the second set holds the second value, and so on. Printing 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 `x = 5` and `y = 10`.

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 [unoptimized + debuginfo] target(s) in 1.50 secs
  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 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 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:

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

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 crates 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 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 [unoptimized + debuginfo] target(s) in 1.50 secs
```

These lines show 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.

Ensuring Reproducible Builds with the `Cargo.lock` File

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

But by default, Cargo will only look for versions greater than 0.3.0 and less 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:

Cargo.toml

```
[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 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

Now that you've added the `rand` crate to *Cargo.toml*, let's start using `rand`. The next step is to update *src/main.rs*, as shown in Listing 2-3.

```
use std::io;
❶ use rand::Rng;

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

    ❷ 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: Adding code to generate a random number

First, we add a use line: `use rand::Rng` ❶. 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 more lines in the middle ❷. 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. 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 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.

NOTE

*You won't just know which traits to use and which functions and methods to call from a crate. 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 middle of 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)
   Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
   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. Note that this code won't compile quite yet, as we will explain.

src/main.rs

```
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 statement ❶, bringing a type called `std::cmp::Ordering` into scope from the standard library. Like `Result`, `Ordering` is another enum, 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`. Then it returns a variant of the `Ordering` enum we brought into scope 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 make sure that you handle them all. These features will be covered in detail in Chapter 6 and Chapter 18, 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. 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, `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 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 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 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 here 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 real number type so we can compare it numerically to the secret number. We can do that by adding the following two lines 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 Rust allows us to *shadow* the previous value of `guess` with a new one. This feature is often used in 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, such as `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. Although `u32` can contain only numerical characters, the user must press `ENTER` 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 result of pressing `ENTER`. 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` mean that Rust will infer that `secret_number` should be a `u32` type 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 as the `read_line` method does (discussed in “Handling Potential Failure with the `Result` Type” on page 17). 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[unoptimized + debuginfo] target(s) in 1.50 secs
  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 that now 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 into a loop from the guess input prompt onward. Be sure to indent the lines inside the loop 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 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 23: 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)
  Finished dev [unoptimized + debuginfo] target(s) in 1.50 secs
  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 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

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 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 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--
```

Listing 2-5: Ignoring a non-number guess and asking for another guess instead of crashing the program

Switching from an `expect` call to a `match` expression is how you generally 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` or `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 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 underscore, `_`, 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 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 [unoptimized + debuginfo] target(s) in 1.50 secs
  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::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 2-6: Complete guessing game code

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, 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.