2 - Programming a Guessing Game

This chapter is going to work with a hands-on project to learn more about Rust. It'll introduce a lot of important concepts such as `let`, `match`, `methods`, `associated functions`, and more.

It'll be a guessing game, where a number is randomly generated, and then a user has to guess what the number is.

Setting up a new project

To set up a new project, go to projects and make a new project with Cargo

```
$ cargo new guessing_game
$ cd guessing_game
```

`cargo new` takes the name of a project as the first argument, and creates a new Cargo directory for it.

Processing a Guess

The first part of this program is going to be asking a user for input and storing a guess, which looks like this :

```
// 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}")
}
```

To start off, we have the line

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

This indicates that we are going to be importing and using the standard Rust library.

Next is the line

```
fn main()
```

The `fn` indicates that this is a function, the rest is pretty standard.

After that we have

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

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

Which looks familiar, but note the `!` at the end of the line. This indicates that we are actually using a `macro` instead of a function, which we'll get into later, but is important to know for now. Next we have

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

So we need to talk about `variables`.

Storing Values with Variables

We can create a variable by using the `let` keyword.

```
let x = 5;
```

Variables are *immutable* by default in Rust, meaning their values cannot be altered.

If we tried to alter the variables value, we would get this error. For example, this code

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

Would give this error.

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:3:5
```

```
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     x += 1;
  |     ^^^^^^ cannot assign twice to immutable variable
```

This error tells us that we are trying to assign a value to an immutable variable that has already been assigned.

In order to actually be able to change the data's value, we use the `mut` keyword.

```
let mut x = 5;
```

With this, we can now mutate the data in the variable freely

```
// main.rs

fn main(){
        let mut x = 5;
        println!("{x}");
        x += 1;
        println!("{x}");
}
```

This will compile fine.

Knowing this, we now know what the next line of code does.

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

It creates a `guess` variable of type `String` that is mutable.

Receiving User Input

Our next piece of code looks like this

```
io::stdin()
        .read_line(&mut guess)
```

This calls the `read_line` method to get user input. We pass the `&mut guess` as a parameter to be the variable we want to store our gathered user input in.

The `&` shows that this is a reference, which is immutable by default, so we write `&mut guess` instead of `&guess`.

Handling Potential Failure with the Result Type

Following this line of code, we have

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

When we read something with `read_line`, it returns a `Result` value, which is an enum with two types: `Ok` and `Err`. Like other value types, `Result` has methods defined in it, one of which is `expect`. If the `read_line` method returns an `Err`, `Expect` lets us handle errors that come up.

Printing Values with `println!` Placeholders

When we want to print the value of a variable, we use `{}` in the print statement

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

This `{}` will read the value of a given variable and print it

```
// main.rs

fn main() {
        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

We can test the first part of the game with

```
$ cargo run
```

Which will build and run the project.

Generating a Secret Number

Next, we'll work on generating a random number. For this we'll use the `rand` crate and generate a number between `1` and `100`.

To add our random functionality, we are going to have to import a Crate. To do this, we go to the `Cargo.toml` file and add this under the `[Dependencies]` header

```
// Cargo.toml

[Dependencies]
rand = "0.8.3"
```

This will import the `rand` crate with a version of `0.8.3`.

Cargo stores all crates in a registry, which tracks all versions of crates.

Once we've imported the crate, we can use the `rand::thread_rng()` function of random with a `gen_range` method to generate a number between `1` and `100`.

```
let secret_number = rand::thread_rng().gen_range(1..=100);
```

Comparing the Guess to the Secret Number

Now that we have user input and a number, we can compare them.

```
// main.rs

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

fn main(){
        // -- Snippit --

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

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

The `Ordering` type is another enum that has three variants : `Less`, `Greater`, and `Equal`. And after that we have the `cmp` method, which compares two values, taking a reference to what we

want to compare with. It then returns an `Ordering` enum with the result.

The `match` method compares two values, in this case it's `guess` and `secret_number`.

A `match` expression is made up of `arms`. An `arm` is a pattern to match against, and the code that should be run if the value given to `match` fits that arm's pattern.

Here's an example of the `match` expression.

If a user has guessed 50 and the number is 38, then when the code compares 50 to 38, `cmp` returns an `Ordering::Greater` enum value.

When the `match` expression is called, it takes the given values and begins working through each of the arms.

When we start working through the arms, it will run the first: `Ordering::Less` which does not match `Ordering::Greater`, so it moves on. It then gets to `Ordering::Greater`, which is the same, so it starts executing the code in the `Ordering::Greater` arm, printing "Too big!" to the screen.

Remember though that one of our numbers `guess` is defined as a String, and the other (secret_number) is defined as an integer, which means we have to convert the string before we compare them. To do this we use

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

The first part of this is

```
guess.trim().parse()
```

When we get a guess from a user, there is a `\n` added at the end of the string so we need to trim it with `.trim()`, so something like `5\n` will just be `5`. We need to tell Rust what value we want to convert to, so we use `let guess: u32`.

Then we have the ".parse()" method. The "parse" method lets us change a string into a different data type.

The `parse` method also only works on convertible numerical characters, and thus has the potential to fail. Because of this, it also returns a `Result` value, which can be handled to execute error handling code. If `parse` can successfully convert a number, it will return `Ok`, and if not, it will return `Err`.

Allowing Multiple Guesses with Looping

The `loop` keyword creates an infinite loop, which we'll use to keep the game running.

```
// --Other code--

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

loop { // Loop declaration
        println!("Please input your guess");

        // --Other code--

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

Here we've moved everything from the guess input into the loop function.

It will now continuously poll the user's input until specified otherwise.

Quitting After a Correct Guess

Now we'll tell the game to quit when the user wins by using a `break` statement.

```
// -- Other code --

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

Adding the break line after the println! makes the program exit the loop.

Handling Invalid Input

We'll also add some error handling for user input validation. Before, we've been crashing the game when the user inputs a non-number, but we don't want to completely crash the game every time this happens. Instead we want to handle the error and simply restore the game to a working point again.

To do this we'll change our `expect` code when we gather our guess from

```
// -- Other code --

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

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

// -- Other code --
```

To

```
// -- Other code --

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

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

// -- Other code --
```

We switch from the `expect` call to a `match` expression to move from crashing the error to handling the error. Since parse returns a Result enum, we can use the match statement to process each of these possible results.

If the `Result` is `Ok`, then the number will be processed as normal. If the `Result` is `Err`, then it will simply continue instead of crashing.

The `_` is a catchall value, which says to match all `Err` values, no matter what information they have.

And with all of this, the guessing game is complete.

Cargo.toml

```
// Cargo.toml
[package]
name = "testing"
version = "0.1.0"
```

```
edition = "2021"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
rand = "0.8.3"
```

main.rs

```
// main.rs

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

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;
            }
        }
```

```
    }
}
```