Rust - Day 2

Programming Guessing Game + Learning Syntax

- Create a new project

```
cargo new guessing_game
cd guessing_game
```

- Write the following code to accept a user input and print it

```
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 Breakdown

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

- Importing the `io` library from the standard library `std`. `io` is used for input/output.
- By default rust has a set of standard library that it brings into the scope of every program and this set is called `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.

```
fn main() {
```

- Defining the main function

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

- Using `println!` macro to print a string to the screen

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

- `let` is used to create the variable.
- **Variables are immutable by default, meaning once we give the variable a value, the value won't change**
- **To make a variable mutable, we add `mut` before the variable name**
- 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`.
- The `::` syntax in the `::new` line indicates that `new` is an associated function of the `String` type.

- **The `let mut guess = String::new();` line has created a mutable variable that is currently bound to a new, empty instance of a `String`.**

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

- `stdin()` is the standard input handle
- `.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.**
- **This would append the user input to the string and not overwrite it.**
- 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.

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

- `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*.
- 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 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.
- 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 this instance of `Result` is an `Ok` value, `expect` will take the return value that `Ok` is holding and return just that value.
- 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
**
```

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

- The `{}` set of curly brackets is a placeholder.
- 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.

```
let x = 5;
let y = 10;
println!("x = {x} and y + 2 = {}", y + 2);
```

- To test the program run - `cargo run`

Generating Random Number

- Rust doesn't have a random number generator in its standard library.
- However, the Rust team does provide a `rand` crate
- **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.**
- To add rand crate, simply add this - `rand = "0.8.5"` line under the `[dependencies]` in the `cargo.toml` file
- **Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.**
- Run `cargo build` to download and compile dependencies from **Crates.io**
- 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.
- Your project will remain at 0.8.5 until you explicitly upgrade.
- **To update all dependencies to new version use - `cargo update`**
- **To explicitely update certain dependencies, just update them in the cargo.toml and run the build again.**

```
use std::io;
use rand::Rng;

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

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

    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}");
}
```

- `use rand::Rng;`. The `Rng` trait defines methods that random number generators implement, and this trait must be in scope for us to use those methods.
- `rand::thread_rng` function 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.
- `gen_range` method on the random number generator. This method is defined by the `Rng` trait.
- `start..=end` means inclusive on the lower and upper bounds.

Comparing the Guess to the secret number

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

fn main() {
```

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

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

    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);

    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 and has the variants `Less`, `Greater`, and `Equal`. These are the three outcomes that are possible when you compare two values.
- The `cmp` method compares two values and can be called on anything that can be compared.
- We use `match` expression 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.***
- *Note: this code won't compile due to some reasons mentioned ahead.**

```
$ cargo build
 Downloading crates ...
  Downloaded rand_core v0.6.2
  Downloaded getrandom v0.2.2
  Downloaded rand_chacha v0.3.0
  Downloaded ppv-lite86 v0.2.10
  Downloaded libc v0.2.86
   Compiling libc v0.2.86
   Compiling getrandom v0.2.2
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.10
   Compiling rand_core v0.6.2
   Compiling rand_chacha v0.3.0
   Compiling rand v0.8.5
   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}`
note: method defined here
  --> /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/cmp.rs:839:8

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

- **Rust has a strong, static type system** - You define the types while defining variables unlike JS or Python
- It can also infer types from the usage when types are not explicitely defined like in this case - `let mut guess = String::new()`
- Some of Rust's number types - **1 and 100: `i32`, a 32-bit number; `u32`, an unsigned 32-bit number; `i64`, a 64-bit number;**
- **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`**
- **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. 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.

Allowing multiple guesses with looping

- `loop` keyword creates an infinite loop.

```
    // --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!");
                break;
            }
        }
    }
}
```

Handling invalid input

```
    // --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--
```

- 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.

- 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.

> That wraps up the Guessing number game.

Final Code

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

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

    let secret_number: u32 = rand::thread_rng().gen_range(1..=100);

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

        let mut guess: String = 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;
            }
        }
    }
}
```

Key Learnings

- Rust has strong, static type system.
- It can infer types of variables from usage if not explicitely defined.
- Variables are defined using `let` keyword
- By default all variables are immutable and `mut` keyword is used to make them mutable.
- Mutable String Variable declaration - `let mut guess = String::new();`
- We use `io::stdin().read_line(&mut guess).expect("Failed to read line");` to read value and store it in `guess` variable and throw error whenever any wrong input is entered.
- `io` library present in the standard library `std` and is used for input/output.
- `println!()` is a macro not a function. ! is used to depict macros.
- `read_line` stores the input into a string and also returns a `Result` value which is an `enum` with two variants - `Ok` and `Err`. `Ok` incidates successful operation and 'Err' contains information on how the operation failed. It causes the program to crash if not handled properly.
- `println!("You guessed: {}", guess);` You can define placeholders for values/expressions in println!() macro.

- Rust doesn't have a random number generator in its standard library but has a `rand` crate.
- Crates are of two types
  - Binary - These are executable crates.
  - Library - These contain code that is intended to be used in other programs.
- `Crates.io` is rust ecosystem to post and get open sourcce crates from.
- `Cargo.lock` file is used to maintain the versions ofthe crates across builds
- To update all dependencies to new version use - `cargo update`
- To explicitely update certain dependencies, just update them in the cargo.toml and run the build again.
- `match` expression is used to compare two varibles and then matches the output against the patterns defined in the body of it.
- `Ordering` is an enum containing values returned by the `cmp` method - Less, Greater, and Equal
- `Shadowing` is a concept that allows reuse of variable names
- `loop` keyword is used to run a block of code infinitely.
- To handle invalid inputs we can use `match` to match the `Result` enum from the input operation and check if it matches the requirement and handle the other cases.