2 - Variables

This chapter is going to go over the most common programming concepts, and show how they are represented in Rust.

Declaring a Variable

As mentioned in the previous chapter, variables in Rust are immutable by default in order to encourage safe coding practices, but of course they can also be made mutable as well.

To start, we'll make a new project `cargo new variables`.

And then add this code

```
// main.rs
// Shows declaring and using a variable

fn main(){
        let x = 5;
        println!("The value of x is {x}");
}
```

```
  The value of x is 5
```

This shows the basic declaration and use of a variable. Now, let's try to change the value in this variable.

```
// main.rs
// Shows a `cannot assign twice to immutable variable` error

fn main(){
        let x = 5;
        println!("The value of x is {x}");
        x += 1;
        println!("The value of x is {x}");
}
```

When we try to run this, we get a compiler error, saying there is an immutability error, why is this? In Rust, variables are *immutable* by default, meaning their values cannot be changed. If we wanted to be able to change their stored values, we have to add the `mut` keyword when we declare it.

Mutable Variables

Let's test that out, adding the `mut` keyword right after `let`.

```
// main.rs
// Showcases mutable variables with `mut` keyword.

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

This lets us mutate the variable how we wanted.

Constants

Constants are similar to variables, in that they are named and have values that can be changed, but there are a few differences.

The first is that you *cannot* use the `mut` keyword with constants, since they are designed to *not be able to change*. Constants can also only be set to the value of a constant expression.

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Here the variable is assigned to the value evaluated by the defined expression.

The standard naming convention for constants in Rust is all caps with underscores separating words.

Constants are valid for the entire program's runtime within their specified scope.

`Const` can also be used in a global scope, while `let` can only be used inside of a function.

```
// main.rs
// Program showcasing a basic const declaration and usage.

fn main(){
        const PI: f32 = 3.14;
        let radius : f32 = 5.0;

        println!("Area of a circle with radius: {radius} is: {}", PI *
(radius * radius));
}
```

Shadowing

You can declare a second variable with the name of another variable that already exists. This is called `Shadowing`. The second variable becomes a shadow of the first, and is the one used whenever the compiler references it from that moment forward. We can shadow a variable by simply using the `let` keyword to define the variable again.

```
// main.rs
// Shows shadowing variables scoped separately

fn main(){
        let x = 5;

        let x = x + 1;

        {
                let x = x * 2;
                println!("The value of x in the inner scope is: {x}");
        }

        println!("The value of x on the outer scope is: {x}");
}
```

This would print this

```
The value of x in the inner scope is: 12
The value of x in the outer scope is: 6
```

Here, the program first defines `x` as `5`. It then creates a new variable, also named `x`, that is the result of `x + 1`, which is `6`. Then we enter our scope where we create another new variable of `x` again, using the old value of `x * 2`.

The `x` in this scope is separate from the `x` outside of the scope, and both have two different values, which we can see from the program printing `12` for the inside scope, and `6` for the outside.

It's notable that *shadowing* is different from `mut`, because it allows us to do transformations to a variable, but still allow it to remain immutable after those changes.

The other notable difference between `mut` and *shadowing* is that shadowing allows us to redefine a variable with a new type.

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

Here, the first instance of spaces is a string, and the second variable is shadowed as a number.

Question 1

Determine whether the program will pass the compiler. If it passes, write the expected output of the program if it were executed.

```
// main.rs

fn main() {
        let mut x: u32 = 1;
        {
                let mut x = x;
                x += 2;
        }

        println!("{x}");
}
```

Answer

This DOES compile. It prints `1`. The statement `x += 2` only affects the shadowed `x` inside the inner curly braces, not the outer `x` on the second line.

Question 2

Determine whether the program will pass the compiler. If it passes, write the expected output of the program if it were executed.

```rust
// main.rs

fn main() {
        let mut x: u32 = 1;
        x = "Hello world";
        println!("{x}");
}
```

Answer

This DOES NOT compile. A variable cannot be assigned to a value of a different type than its original type. Here, we declare `x` first, explicitly as a `u32` type, and then we try to assign a `String` type to it, which is not allowed.