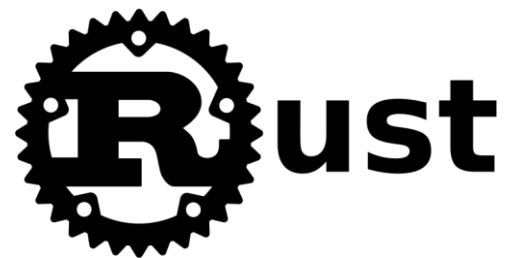


Introduction to Rust



Chapter 4



ROADMAP

3

Basic Tool Installation



Using Cargo and Crates



Conditionals and Loops



Your first lines of Rust

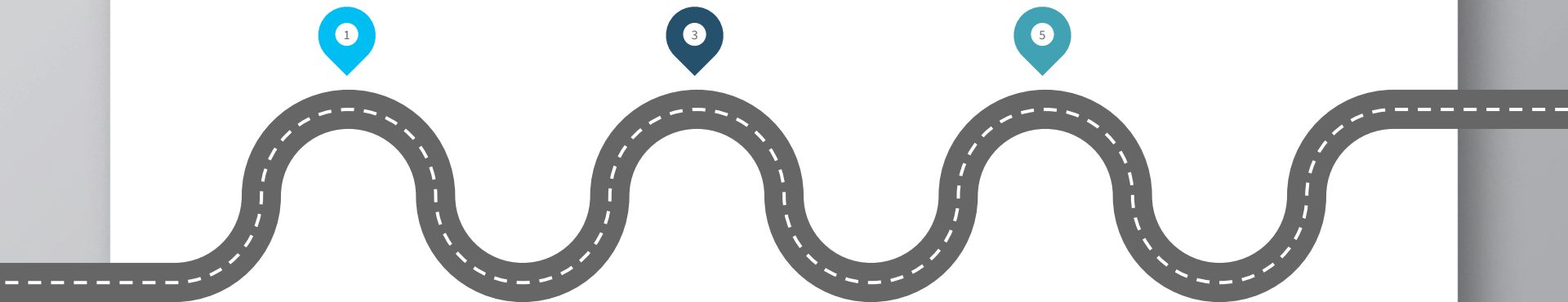
You are here



Data Types and User Input



Project



4.

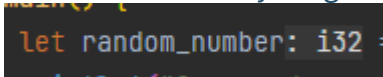
Data types and user input

In the previous lecture when we declared the variable “random_number” we didn’t specify what type that variable would be. We let the Rust compiler to deduce the type on its own, but in most situations it is required of us to declare what type a variable is going to be before compilation.

In order to do that we write after the variable “: <type>”, for example: **let random_number : i32** = ... Here we declare the variable random_number as an integer of 32 bits.

That is exactly what the Rust compiler automatically assigned to our variable, but we could also declare it as an unsigned integer (**u32**), meaning no negative numbers as there was no possible negative number assigned to this variable in our specific example.

The extension of Visual Studio Code rust-analyzer gave us a hint on how the compiler would interpret the variable “random_number” here it shows us that the variable will be of type i32.



```
let random_number: i32 =
```

Data types

6

Integer types:

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Rust Data Types:

Floating types: **f32**, **f64** (e.g. 6.4, 3.14)

Boolean type: **bool** (true/false)

Character type: **char** (e.g. 'a', 'z')

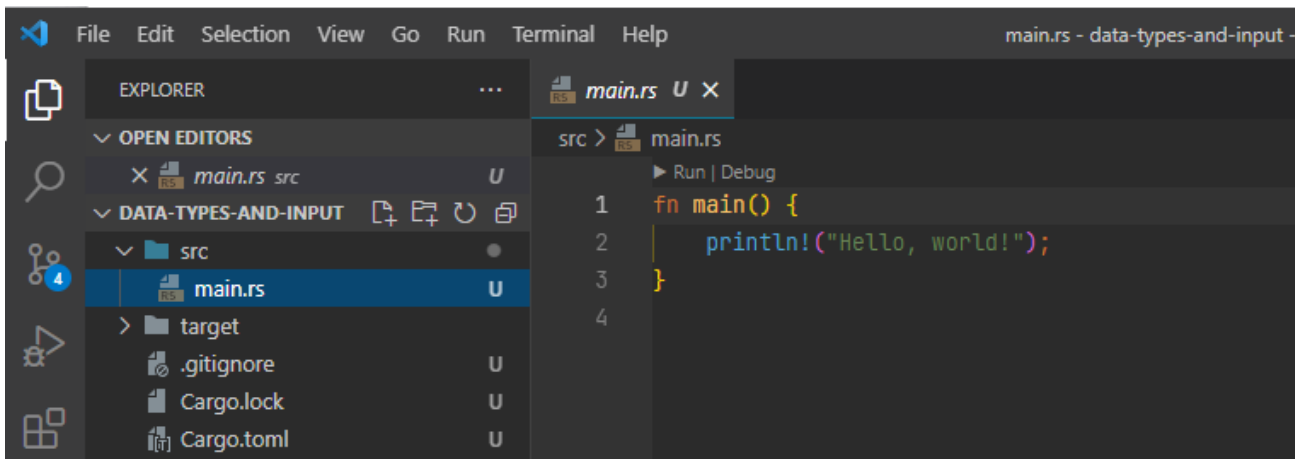
String object: **String** (e.g. "hello")

There is also the String slice type: **&str** which relates to the String object but those are more advanced and will be discussed in a different lecture.

Data types and user input

7

Let's create a new Cargo project for this lecture with the name "data-types-and-input". As a reminder we go to our main "rust-course" folder and use the command "cargo new data-types-and-input" and then open the new folder in Visual Studio Code.



Data types and user input

8

Let's create a variable of type `u8` (unsigned integer of size 8 bits which means a number between 0 and 255) and let the user input its value.

In order to read from user input we have to **use `std::io`**; (standard input output) so let's add this line on the top of our file, above the main function. `std` is the Standard Library crate available for all Rust projects by default so we do not need to add anything in our dependencies section of the `Cargo.toml` file.

First we have to discuss a bit about mutability. By default every variable in Rust is immutable, meaning its value cannot be changed. In order to have a mutable variable we have to declare it as such using the **`mut`** keyword. e.g. **`let mut number`**...

For now we will declare `number` as a new `String` in order to let the user set it to some characters and then we will convert it to a number and assign it to a `u8` variable.

Let's add inside our main function: **`let mut number = String::new();`**

Now we are ready to read from the user input and assign it to the variable `number`.

In order to read user input we are going to use the `read_line()` method of the `io::stdin()` :

```
io::stdin().read_line(&mut number);
```

The `read_line` method takes as argument the variable we want to store the input in.

Here we use **&mut number** which is read as “a mutable borrow of the variable number”. Mutable because of the keyword `mut` which means we can change the variable’s value and borrow because of the `&` character which means we will take the variable `number` just to change its value and then give it back to the main function. Borrowing is a more advanced concept of Rust for a later lecture.

If you run the program now there will be an error! That’s because Rust doesn’t allow us to run code that has not checked for errors, and that’s one of the reasons Rust is so safe compared to C/C++. Here there can be an error when reading user input so we have to use the **.expect** method and give an error message as an argument for the possibility of an error when reading from user input. So our final line of code is:

```
io::stdin().read_line(&mut number).expect(“Failed to read line”);
```

We can now test our code by printing what the user entered. Let's use the `println!` macro to do that.

`println!("{}", number);`

```
src > main.rs > ...
1 use std::io;
2
3 ▶ Run | Debug
4 fn main() {
5     let mut number: String = String::new();
6     io::stdin().stdin
7         .read_line(buf: &mut number) Result<usize, Error>
8         .expect(msg: "Failed to read line");
9     println!("{}", number);
10 }
```

By using **cargo run** we can enter a line and it will be repeated to us, but as you can already see that's not exactly what we wanted, we wanted the variable `number` to be of type `u8`. In order to do that we can use the methods **`trim()`** which deletes spaces/tabs and new lines from a `String`, and **`parse()`** which converts a `String` into a number, on our variable. We will set the content of `number` on the same variable but this time not as a string but as a number. This is called “shadowing” in Rust and it goes like this:

```
let number: u8 = number.trim().parse();
```

As before there might be an error that we haven't handled, for example inputting a string cannot be converted to a number, or inputting a number outside of the range 0-255 that `u8` expects, so we have to handle that:

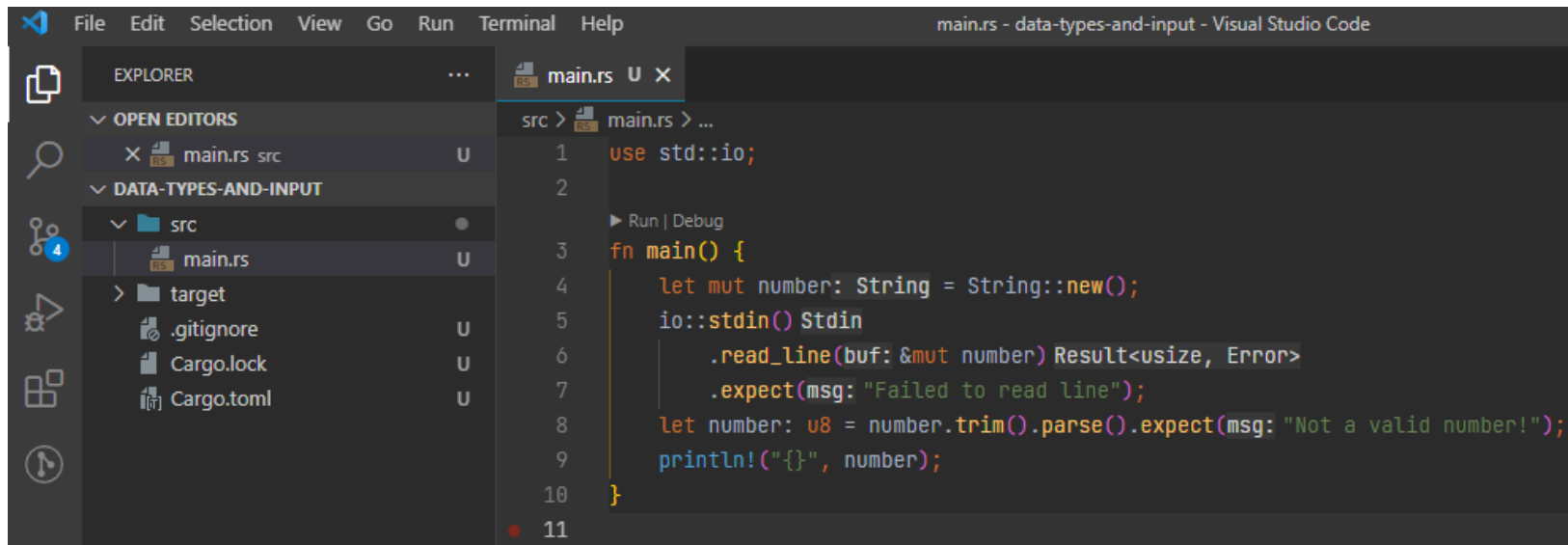
```
let number: u8 = number.trim().parse().expect("Not a valid number!");
```

```
let number: u8 = number.trim().parse().expect(msg: "Not a valid number!");
```

Data types and user input

12

Our finished program should look something like this:



The screenshot shows the Visual Studio Code interface with a Rust project. The Explorer sidebar on the left shows the project structure: a 'src' directory containing 'main.rs', and a 'target' directory. The main editor displays the code for 'main.rs'.

```
src > main.rs > ...
1 use std::io;
2
3 fn main() {
4     let mut number: String = String::new();
5     io::stdin().Stdin
6         .read_line(buf: &mut number) Result<usize, Error>
7         .expect(msg: "Failed to read line");
8     let number: u8 = number.trim().parse().expect(msg: "Not a valid number!");
9     println!("{}", number);
10 }
11
```

**Remember to handle
possible errors!**

THANKS!

14

Any questions?

You can find me at:

» vlasopoulos.v@gmail.com

