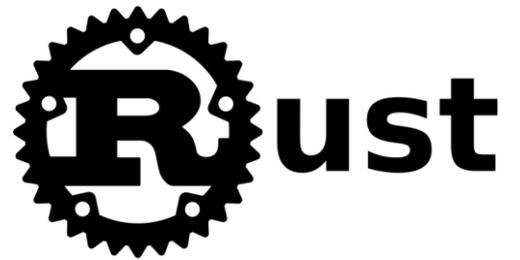


Introduction to Rust



Chapter 3



ROADMAP

3

Basic Tool Installation



Using Cargo and Crates

You are here



Conditionals and Loops



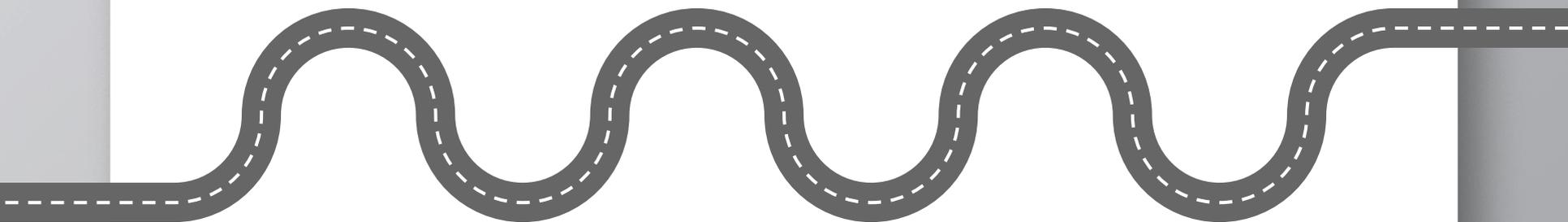
Your first lines of Rust



Data Types and User Input



Project



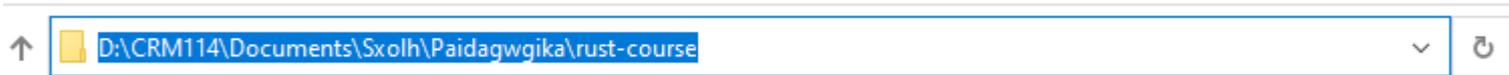
3.

Using cargo and crates

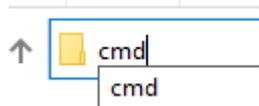
Cargo is Rust’s build system and package manager. From now on when creating/testing/running a rust project we will use Cargo. Cargo is already installed in your system as it comes packaged with the Rust installer.

Let’s use the cargo command to make a new cargo project for this lecture. First you will need to open a command line in our “rust-course” folder. You can use **cd** if you’re familiar with it or you can just highlight the path on the folder:

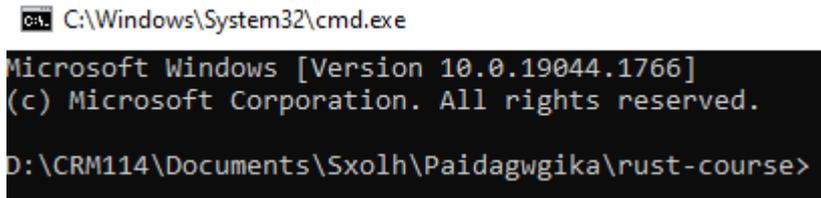
me share view



Type “cmd” and press enter:



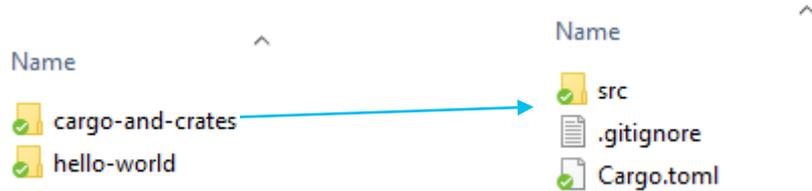
A new terminal will open in the highlighted folder:



Now lets use the cargo command. To create a new project we can use the command “cargo new <project name>” so lets create a new project called cargo-and-crates with “**cargo new cargo-and-crates**” and press enter.

```
D:\CRM114\Documents\Sxolh\Paidagwgika\rust-course>cargo new cargo-and-crates
Created binary (application) `cargo-and-crates` package
```

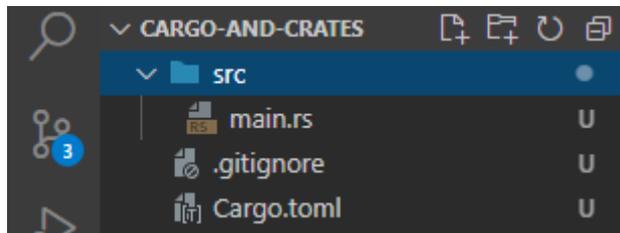
Cargo has created a new folder for us with the name we gave it and added some files inside it.



For now we can ignore the “.gitignore” file. Lets focus instead on the “src” folder and the “Cargo.toml” file. The “src” folder contains all our source code (*.rs) files. Cargo has created one rust file inside the folder “src” called “main.rs” which contains the Hello World program we created in the previous lecture!

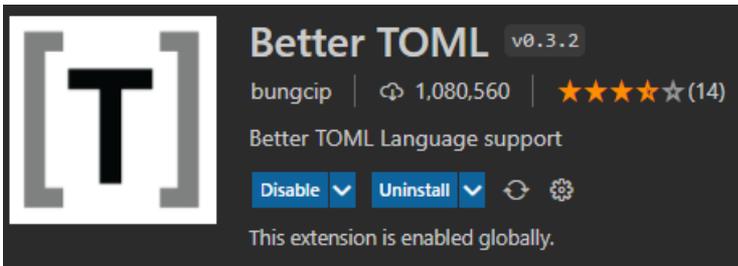
The “Cargo.toml” file is a special file that contains all of our project’s configurations.

Let’s open the project we just created with Cargo in Visual Studio Code (like last time by opening the folder).

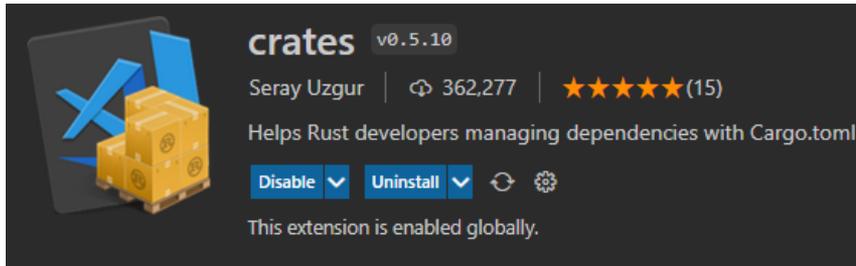


In order to have a better experience with Cargo we are going to add two new extensions in Visual Studio Code.

Better TOML and crates



 **Better TOML** v0.3.2
bungcip | 1,080,560 | ★★★★★ (14)
Better TOML Language support
Disable Uninstall Refresh Settings
This extension is enabled globally.



 **crates** v0.5.10
Seray Uzgur | 362,277 | ★★★★★ (15)
Helps Rust developers managing dependencies with Cargo.toml.
Disable Uninstall Refresh Settings
This extension is enabled globally.

Now that we have the Better TOML extension lets double click Cargo.toml in order to inspect it.

```
Cargo.toml
1  [package]
2  name = "cargo-and-crates"
3  version = "0.1.0"
4  edition = "2021"
5
6  # See more keys and their definitions in Cargo.toml
7
8  [dependencies]
9
```

As we can see it has two sections, [package] and [dependencies].

Under [package] we have the name of our project (which will also be the name of our executable after compilation), the version which defaults to “0.1.0” and the edition of Rust we are using to compile the project with, in this case “2021”.

Under [dependencies] we have nothing (for now). Dependencies are other people’s code libraries that we can use in our program called **crates**.

Let’s run our program once using the cargo command before we move into crates.

Using the terminal we can execute cargo commands. We can just compile our program with the command “**cargo build**” and we can compile and run it with the command “**cargo run**”. Let’s use cargo run:

```
PS D:\CRM114\Documents\Sxolh\Paidagwgika\rust-course\cargo-and-crates> cargo run
Compiling cargo-and-crates v0.1.0 (D:\CRM114\Documents\Sxolh\Paidagwgika\rust-course\cargo-and-crates)
Finished dev [unoptimized + debuginfo] target(s) in 1.05s
Running `target\debug\cargo-and-crates.exe`
Hello, world!
```

The cargo steps start with green text. First it **compiled** our program, then showed compilation information when the compilation ended, and after that it **run** our program, printing “Hello, world!” in the terminal. Under the hood Cargo used the **rustc** command we also used in our first program, but from now on we will let Cargo deal with rustc.

crates.io

As said before crates are other people's libraries of code we can use in our programs. You can find a list of available crates in the crates.io website. In order to use a crate we have to declare it in our Cargo.toml file under [dependencies]. Let's see an example of that using the **rand** crate which gives us the functionality of generating random numbers. Add in the Cargo.toml file under dependencies the line **rand = "0.8.5"**

```
8 [dependencies]
9 rand = "0.8.5"
```

rand is the name of the crate and after the equals we put in quotes the version of the crate we want to use (in this case 0.8.5).

Now run “**cargo build**” and watch the output!

Cargo downloads the rand crate and every other crate needed by rand automatically!

Now let's open our "src/main.rs" file to use the rand crate. Before our main function we will add the line "use rand::Rng;" This will allow us to generate random numbers using the rand crate.

```
src > main.rs > ...  
1 use rand::Rng;  
2
```

Now inside our main function we can create a variable using the **let** keyword followed by the name of the variable. Let's name our variable "random_number". We will use the **gen_range** method of the rand crate to give our variable a value:

```
let random_number = rand::thread_rng().gen_range(1..10);
```

From the rand crate

Call the random number generator

Generate a random number within the following range

The range of numbers from 1 to 10 **not** including 10, so 1,2,3,...,9

Now our `random_number` variable has a random value of 1 through 9. In order to print this number to the console we will use the `println!` macro again but with a twist!

Instead of “Hello, world” inside the parentheses of `println!` we will put “{}” followed by a comma followed by the variable’s name, so:

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

This means that whenever the **placeholder {}** brackets exist they will be replaced by the value of the variable after the comma. You can add more characters within the quotes if you want:

```
println!("Our random number is {}", random_number);
```

Now let’s use **cargo run** to compile and run our program and inspect the output.

```
Our random number is 9
```

Run it a few times to see different random numbers!

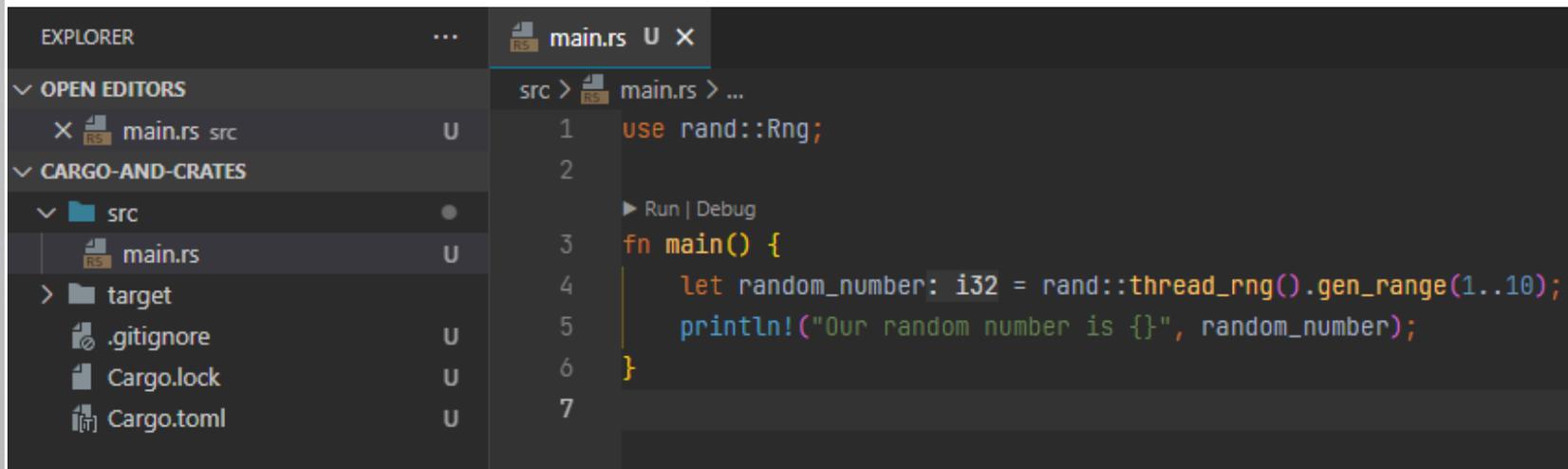
```
Our random number is 4
```

```
Our random number is 6
```

Final product

14

Our final program should look something like this in Visual Studio Code:



The screenshot shows the Visual Studio Code interface with a dark theme. On the left, the Explorer sidebar shows a project structure with a 'src' folder containing 'main.rs'. The main editor window displays the code for 'main.rs' with line numbers 1 through 7. The code includes a 'use' statement for 'rand::Rng', a 'main' function, and a call to 'rand::thread_rng().gen_range(1..10)' to generate a random number, which is then printed to the console.

```
1 use rand::Rng;
2
3 ▶ Run | Debug
4 fn main() {
5     let random_number: i32 = rand::thread_rng().gen_range(1..10);
6     println!("Our random number is {}", random_number);
7 }
```

```
use rand::Rng;
```

THANKS!

16

Any questions?

You can find me at:

» vlasopoulos.v@gmail.com

