Rust Setup

Installation, Compiling, Execution, Testing

author nlahn@radford.edu; modified by ibarland@radford.edu.

First homework exercise

- Check D2L! Due soon!
- You will learn everything you need for it today / next class
- Do it ASAP in case you encounter installation issues
- If you encounter issues, contact me
  - ibarland@radford.edu

Book Reading

- Rust Book Ch. 1 : Getting started
  - https://doc.rust-lang.org/book/ch01-00-getting-started.html

Command Line Installation

- We will be using command line in this class
  - Specifically, you need to know some basic Linux command line
- If you are using Mac / Linux, you already have a command line installed
- If you are using Windows, you should have Powershell
  - Not all command are the same as Linux, but they are similar
  - Please install a 3rd party Linux command line emulator, such as Cygwin, so that you can follow along with any commands I use
    - https://www.cygwin.com/

Rust Installation

- The following link provides an installer for Rust on Windows, as well as some additional instruction
  - https://www.rust-lang.org/tools/install
- If you are on Mac or Linux, the same link above should work.
  - Your OS should automatically be detected by the site, and you will get specific command line instructions to run

Playground

- If you want to test something out, but don’t have your Rust installation configured yet, try this:
  - https://play.rust-lang.org/?version=stable&mode=debug&edition=2021
- The playground should not be used as a substitute for a working Rust installation. It’s just meant for quick and simple testing.

IDE : VSCode

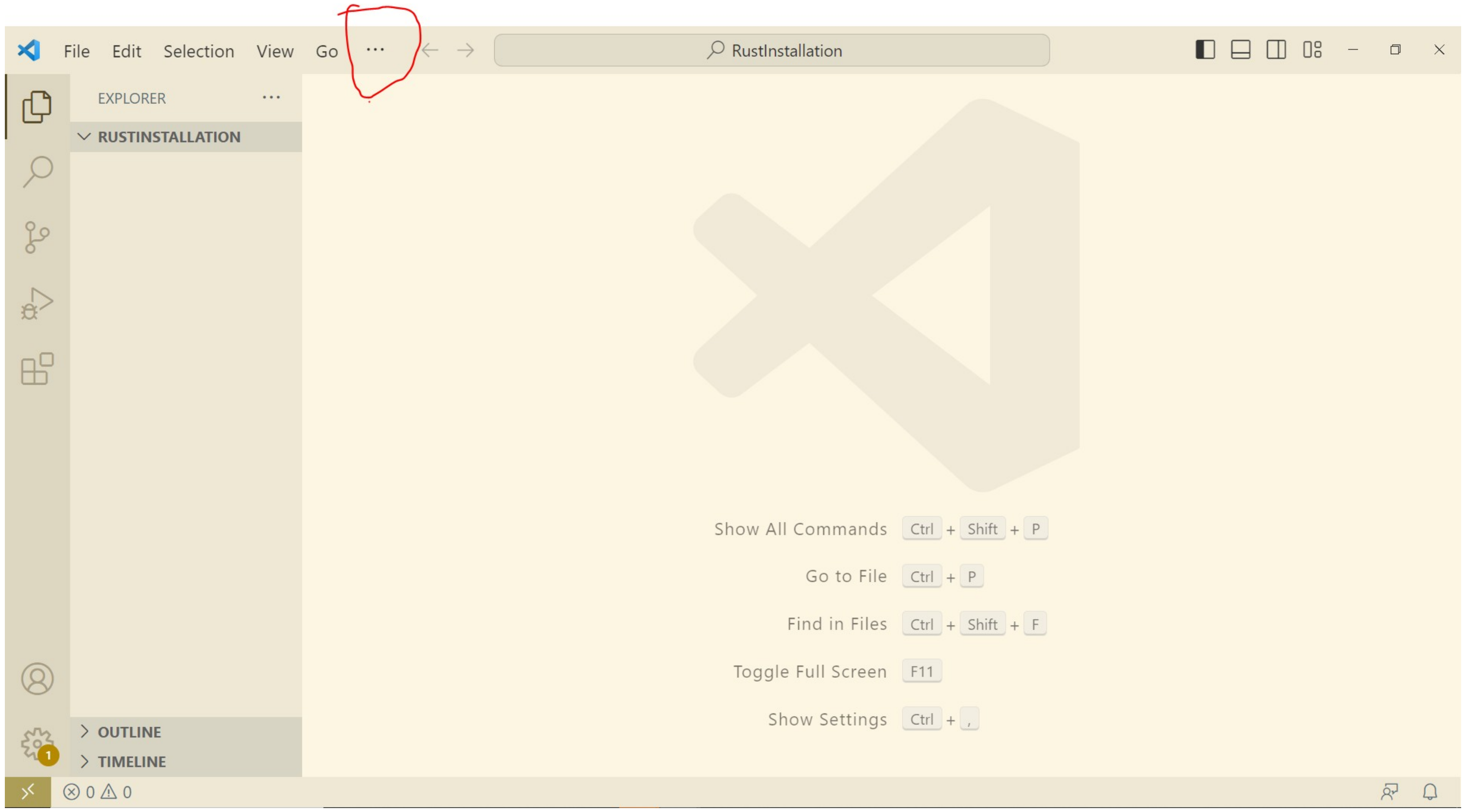
- Rust does not have a good designated IDE (that I know of)
  - Update 2024 – you can try “Rust Rover” by Jetbrains!
- However, it does have a *really good* plugin for the VSCode IDE
- You should *really really get it*
- To install VSCode, go here:
  - https://code.visualstudio.com/download
- For detailed instructions on how to install the Rust plugin, see here:
  - https://code.visualstudio.com/docs/languages/rust
- This should be available on any OS!

First Rust Project

- First, make a workspace folder, to house your Rust projects.
  - Do this using the File Explorer on your OS, or command line
- In VSCode
  - Click File -> Open Folder
  - Select the folder for your workspace
- Initially, you will have a blank folder, but we will populate it!

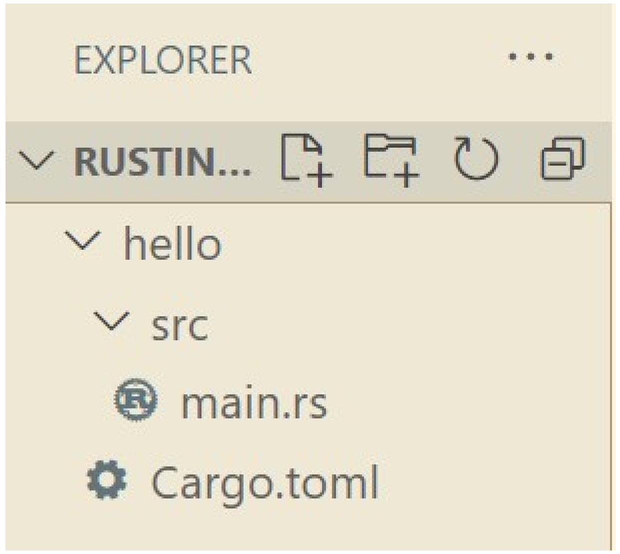
Open Terminal In VSCode

- You might have to click the … at the top of the screen to find the terminal
- Ctrl + Shift + ` works on Windows



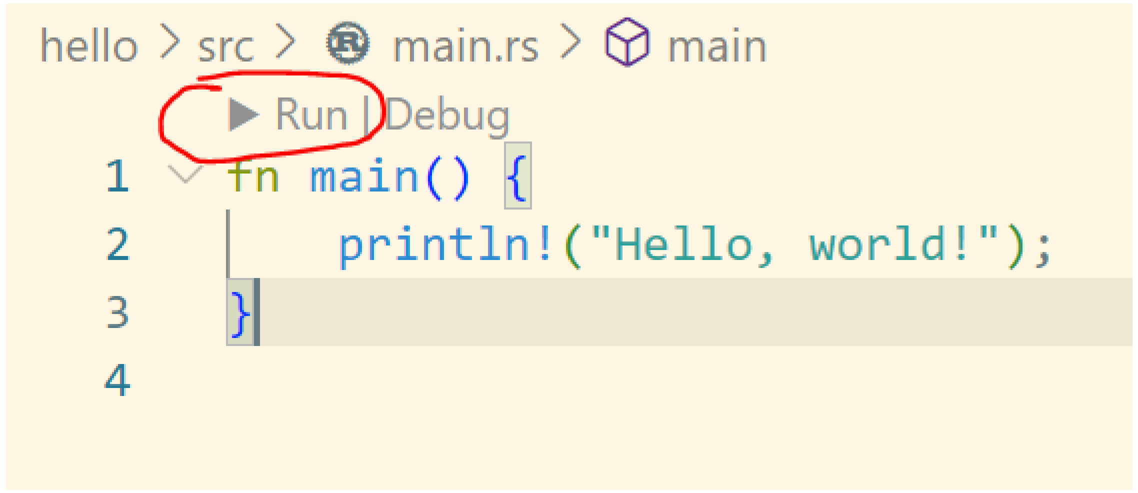
Create Rust Project

- Note : Your terminal should by default be in your project directory
- Run the following command in your terminal
  - cargo new hello
- This will create a few files:
  - A src (source code) directory, with a main.rs file
  - A Cargo.toml file, which contains meta-data about the project



Open main.rs

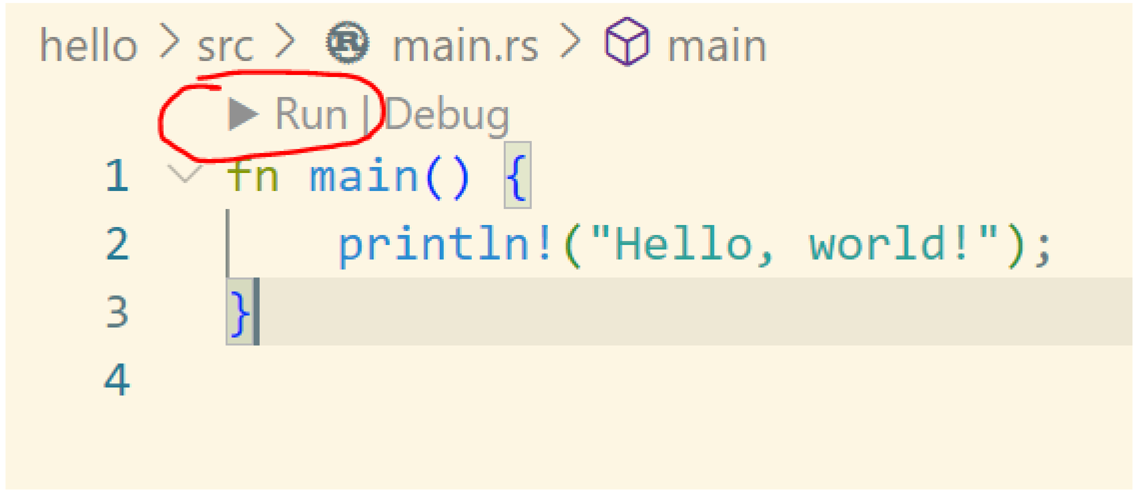
- You will see a hello world program, already there for you
- Run the program:



```
fn main() {
    println!("Hello, world!");
}
```

What happens when you press Run?

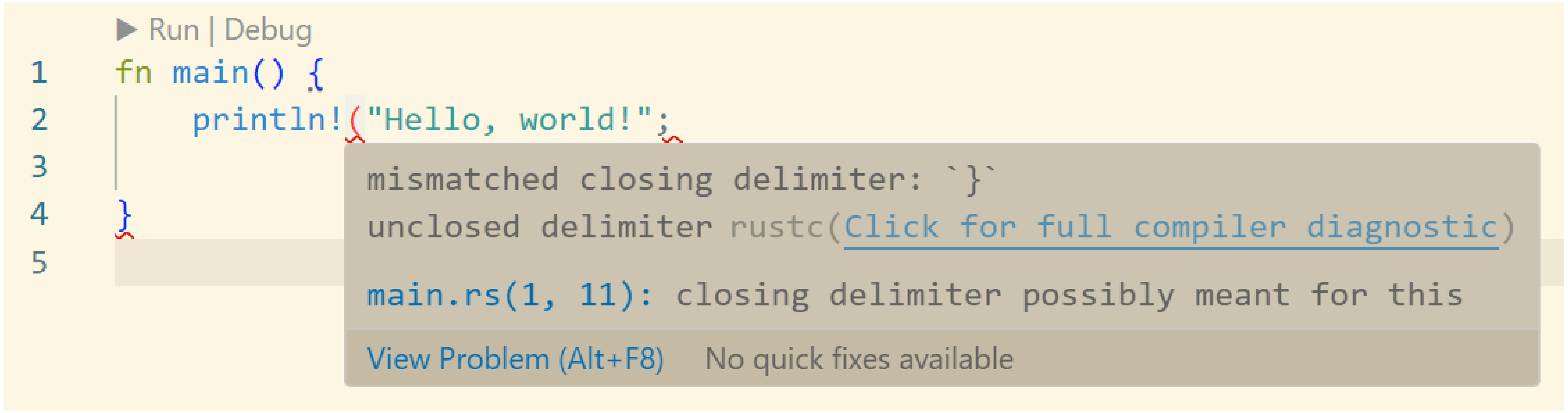
- First, the program is compiled
- This generates a “target” directory, which contains binary machine code
- Then, the program runs the machine code executable



```
fn main() {
    println!("Hello, world!");
}
```

Compile Errors

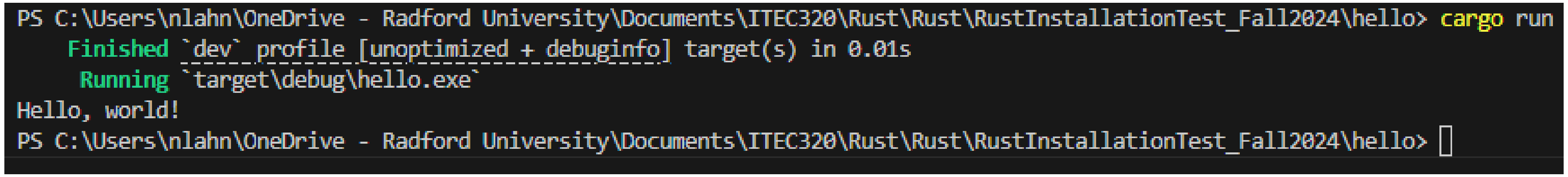
- Try removing a semi-colon from the hello world program
- You should get some errors, highlighted in red
- This is probably similar to your Java IDE



```
fn main() {
    println!("Hello, world!";

}

```

```
mismatched closing delimiter: `}`
unclosed delimiter rustc(Click for full compiler diagnostic)

main.rs(1, 11): closing delimiter possibly meant for this
```

Running the program from command line

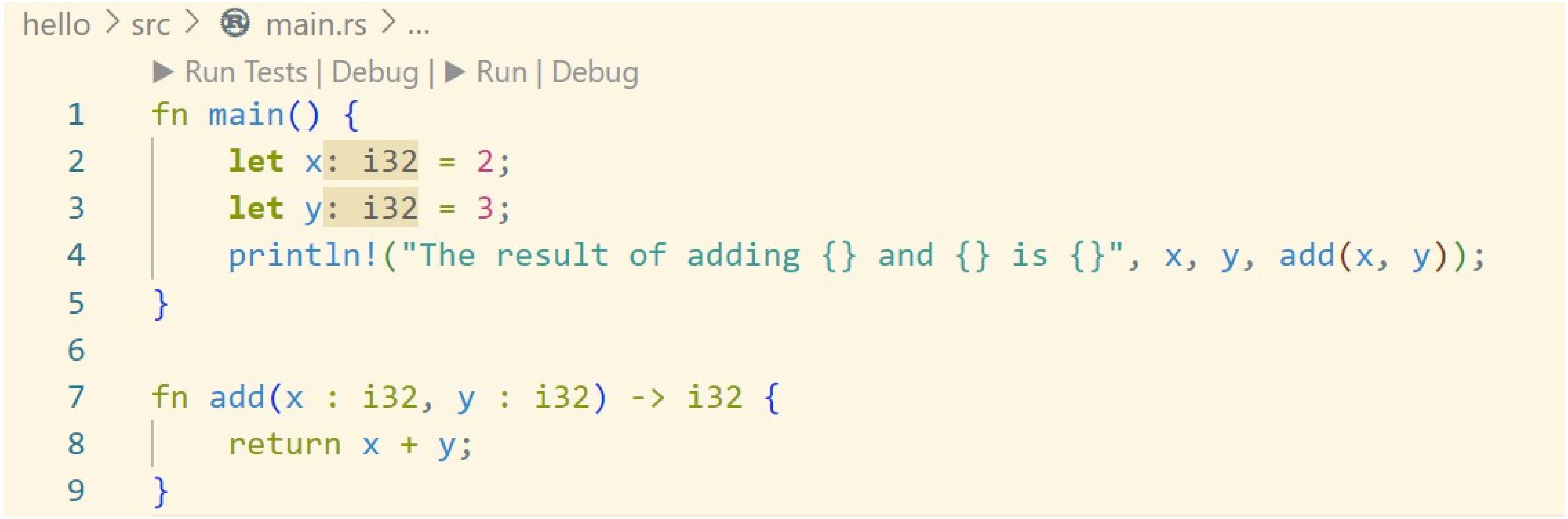


```
PS C:\Users\nlahn\OneDrive - Radford University\Documents\ITEC320\Rust\Rust\RustInstallationTest_Fall2024\hello> cargo run
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\hello.exe`
Hello, world!
PS C:\Users\nlahn\OneDrive - Radford University\Documents\ITEC320\Rust\Rust\RustInstallationTest_Fall2024\hello>
```

A (slightly) more complex program

- Let’s write a program that has:
  - An additional function called “add”, which accepts two integers and returns their sum
  - A main function that:
    - Declares two integers x and y
    - Calls add to get the result of adding them together
    - Prints x, y and the result of adding them, using descriptive output

Solution


hello > src > main.rs > ...

▶ Run Tests | Debug | ▶ Run | Debug

```rust
fn main() {
    let x: i32 = 2;
    let y: i32 = 3;
    println!("The result of adding {} and {} is {}", x, y, add(x, y));
}

fn add(x : i32, y : i32) -> i32 {
    return x + y;
}
```

Writing Tests

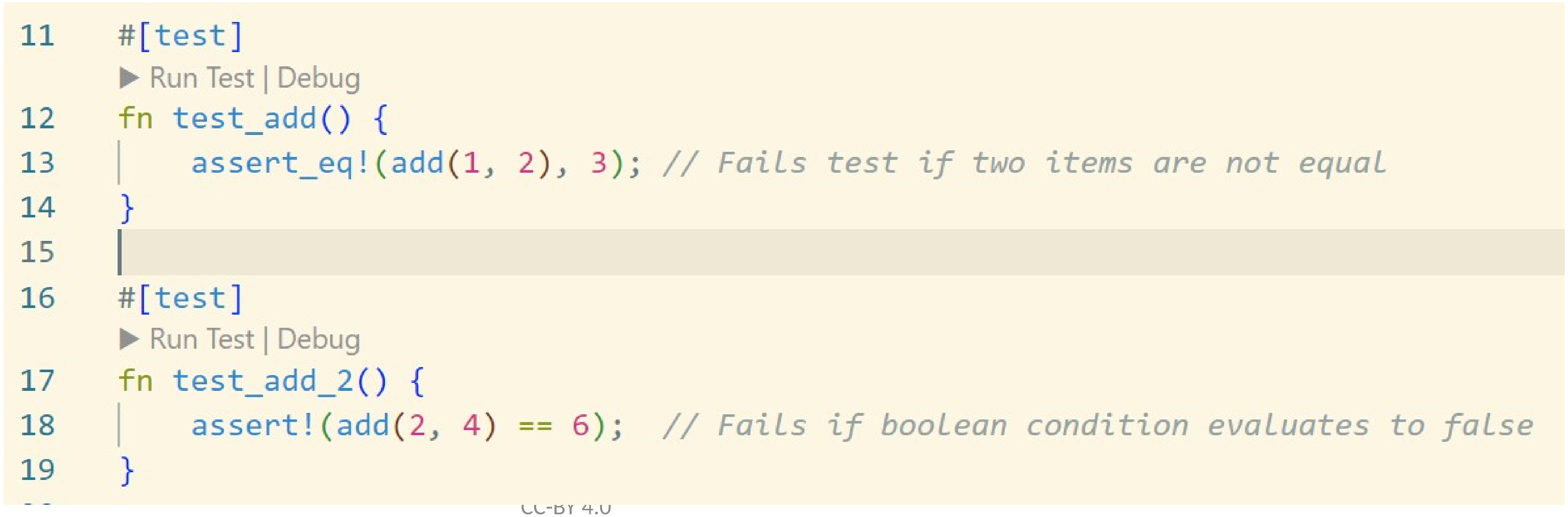
- Writing unit tests is *very important*
  - *Why?*

Writing Tests

- Writing unit tests is *very important*:
  - Obviously you have to test at *some point*. Otherwise, how do you know it works?
  - You could test manually, but:
    - It is very time consuming
    - If you change your program, then you have to, again, rerun all your tests manually
    - If your test fails, it’s kind of difficult to tell where it failed, or reproduce it
- The solution to this problem is *unit testing*

Writing Unit Tests in Rust

- *Any* function in Rust, *anywhere* can be made into a test
- Just add the following annotation:
  - #[test]



```
#[test]
▶ Run Test | Debug
fn test_add() {
    assert_eq!(add(1, 2), 3); // Fails test if two items are not equal
}

#[test]
▶ Run Test | Debug
fn test_add_2() {
    assert!(add(2, 4) == 6);  // Fails if boolean condition evaluates to false
}
```

Running tests

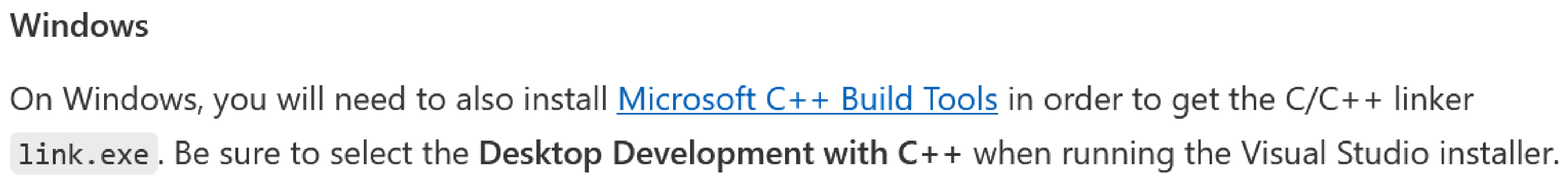
- You can run tests directly from VSCode, by clicking: *Run tests*
- You can also run a single test at a time.
- You can also run tests from command line, using the command
  - `cargo test`
  - (You must be in the Rust project directory, where the .toml is)

Test Everything, Often

- Unlike some other language…. Rust unit testing is *easy*
- *So you have no excuse not to do it*
- For every nontrivial function, put a test right below it
  - Write the test **before** your write the function
  - After you write the function, test it immediately
- When you complete a large section of code:
  - Write tests that ensure the different sub-parts of the code are all working together (integration testing)
- When you change anything at all in your code
  - Rerun all the tests!
- Early error detection → Easier to fix errors → more efficient coding

Debugger installation

- Speaking of more efficient debugging…
  - You should really learn how to use a debugger if you haven't already.
- You can find additional instructions for installing the debugger here:
  - https://code.visualstudio.com/docs/languages/rust#_debugging
- Pay special attention to this part:

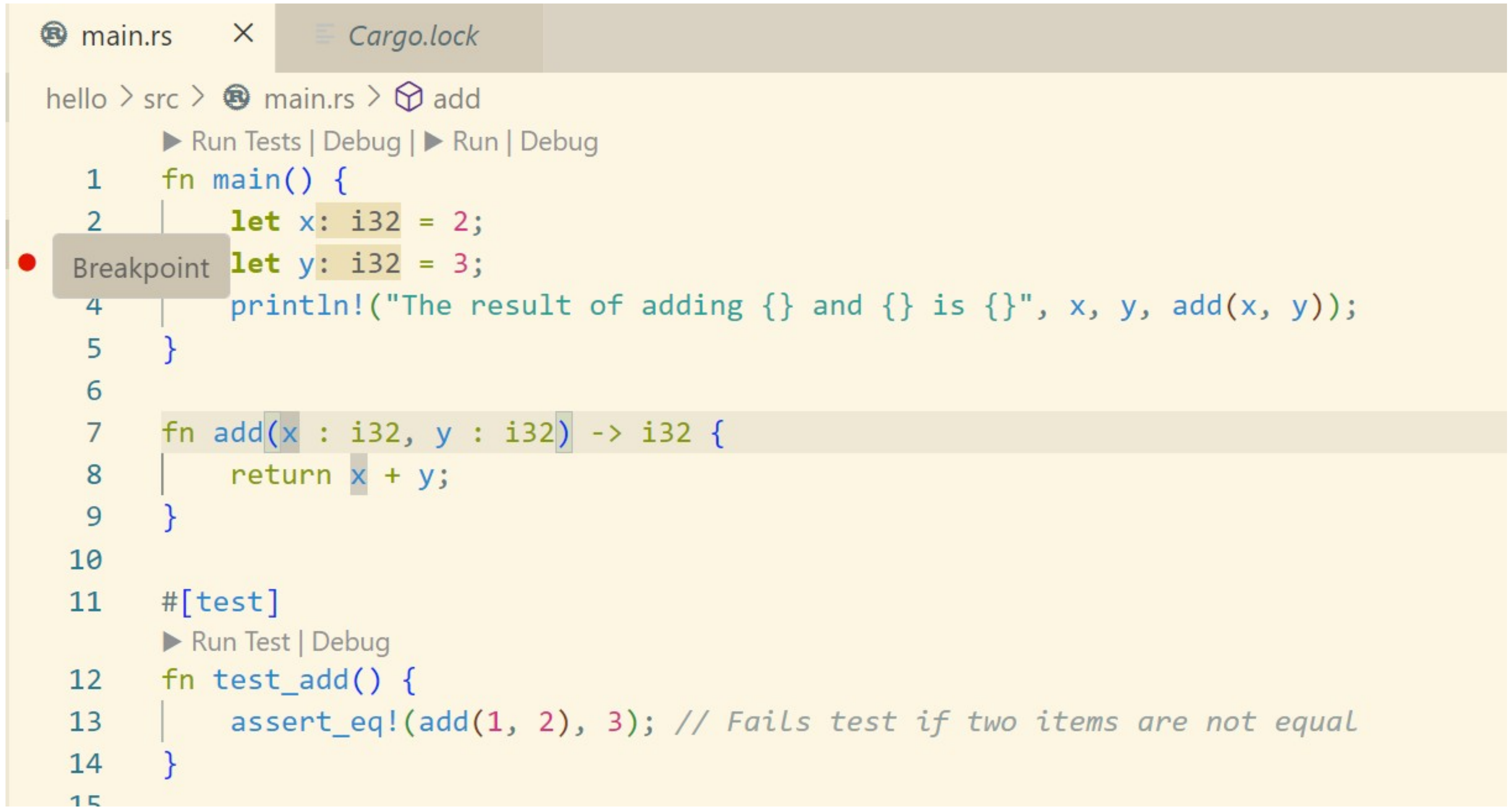


Windows

On Windows, you will need to also install Microsoft C++ Build Tools in order to get the C/C++ linker `link.exe`. Be sure to select the **Desktop Development with C++** when running the Visual Studio installer.

Using the Debugger!

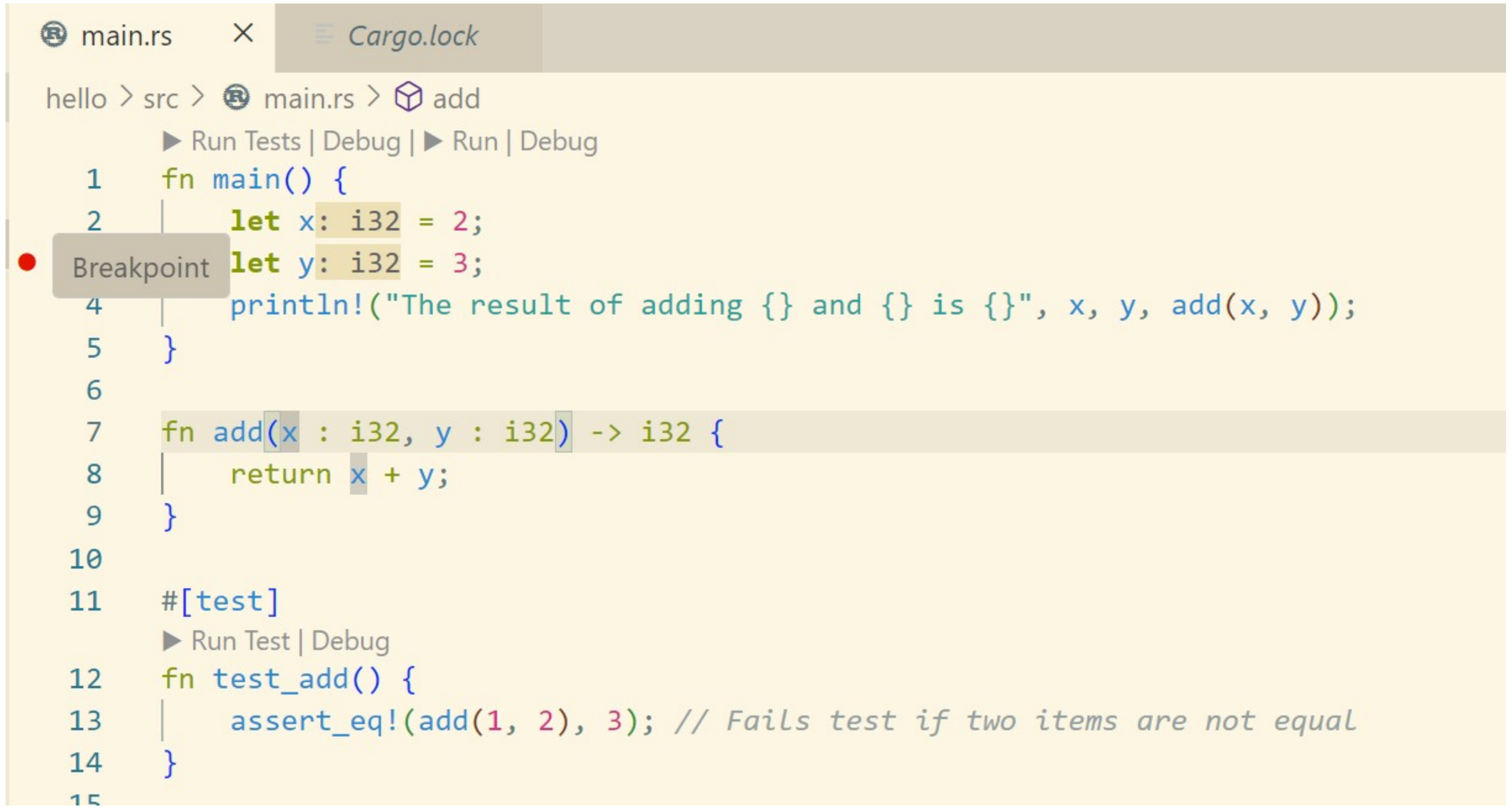
- First, add a "breakpoint" by clicking to the left of some line number


main.rs
Cargo.lock
hello > src > main.rs > add
Breakpoint
▶ Run Tests | Debug | ▶ Run | Debug
▶ Run Test | Debug

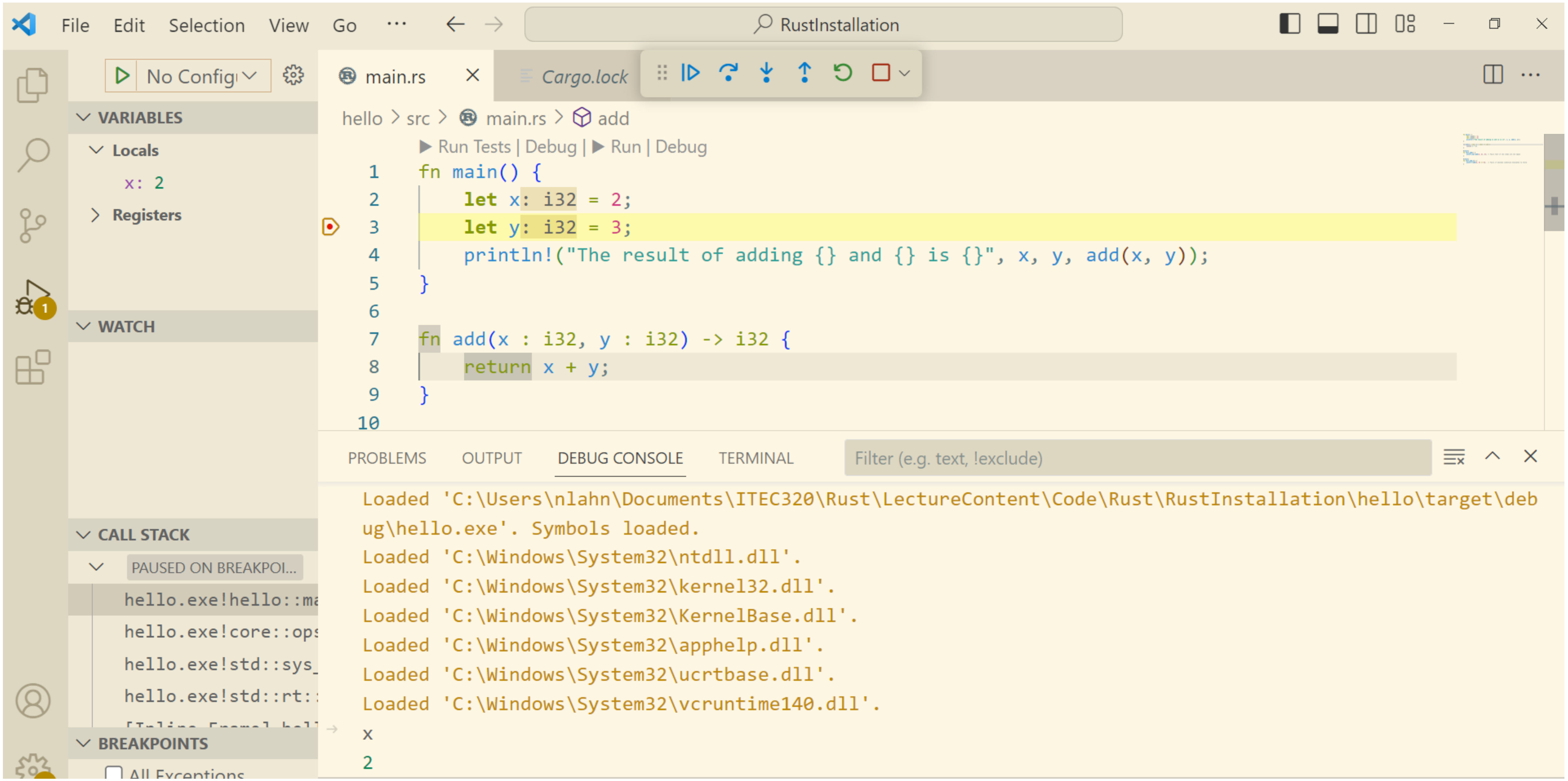
```
fn main() {
    let x: i32 = 2;
    let y: i32 = 3;
    println!("The result of adding {} and {} is {}", x, y, add(x, y));
}

fn add(x : i32, y : i32) -> i32 {
    return x + y;
}

#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3); // Fails test if two items are not equal
}
```

Using the Debugger!

- A breakpoint is a place where the program will stop when you debug it
- Try clicking “Debug” in VsCode, instead of “Run”
  - What happens?


main.rs
Cargo.lock
hello > src > main.rs > add
▶ Run Tests | Debug | ▶ Run | Debug
Breakpoint
▶ Run Test | Debug

```
fn main() {
    let x: i32 = 2;
    let y: i32 = 3;
    println!("The result of adding {} and {} is {}", x, y, add(x, y));
}

fn add(x : i32, y : i32) -> i32 {
    return x + y;
}

#[test]
fn test_add() {
    assert_eq!(add(1, 2), 3); // Fails test if two items are not equal
}
```



```rust
fn main() {
    let x: i32 = 2;
    let y: i32 = 3;
    println!("The result of adding {} and {} is {}", x, y, add(x, y));
}

fn add(x : i32, y : i32) -> i32 {
    return x + y;
}
```

```
Loaded 'C:\Users\nlahn\Documents\ITEC320\Rust\LectureContent\Code\Rust\RustInstallation\hello\target\debug\hello.exe'. Symbols loaded.
Loaded 'C:\Windows\System32\ntdll.dll'.
Loaded 'C:\Windows\System32\kernel32.dll'.
Loaded 'C:\Windows\System32\KernelBase.dll'.
Loaded 'C:\Windows\System32\apphelp.dll'.
Loaded 'C:\Windows\System32\ucrtbase.dll'.
Loaded 'C:\Windows\System32\vcruntime140.dll'.
x
2
```

Some things you can do with the debugger

- View the values of all variables currently visible to the current line of code
- Step forward one line at a time
- Skip to the next breakpoint
- Go inside of functions
- See the value of any expression

- You can debug tests separately from the main program