C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 9: Rust intro & typing

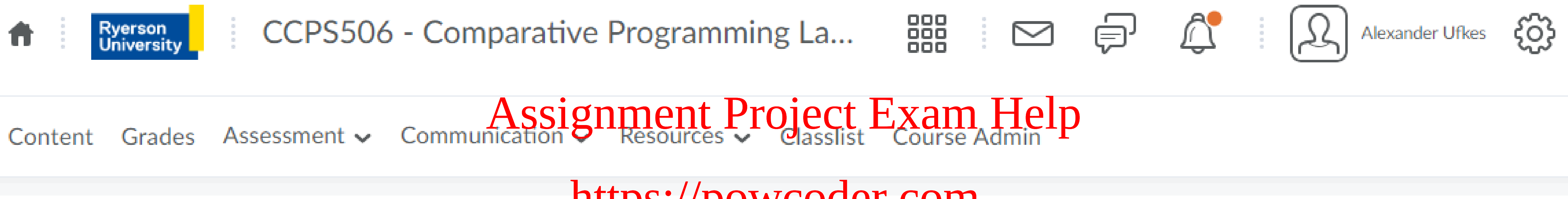
Ryerson University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this original work is held by Alex Ufkes. Students registered in course CCPS 506 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration



- Getting closer! Rust is our last language.
- Don’t forget about the assignments!

Moving on…

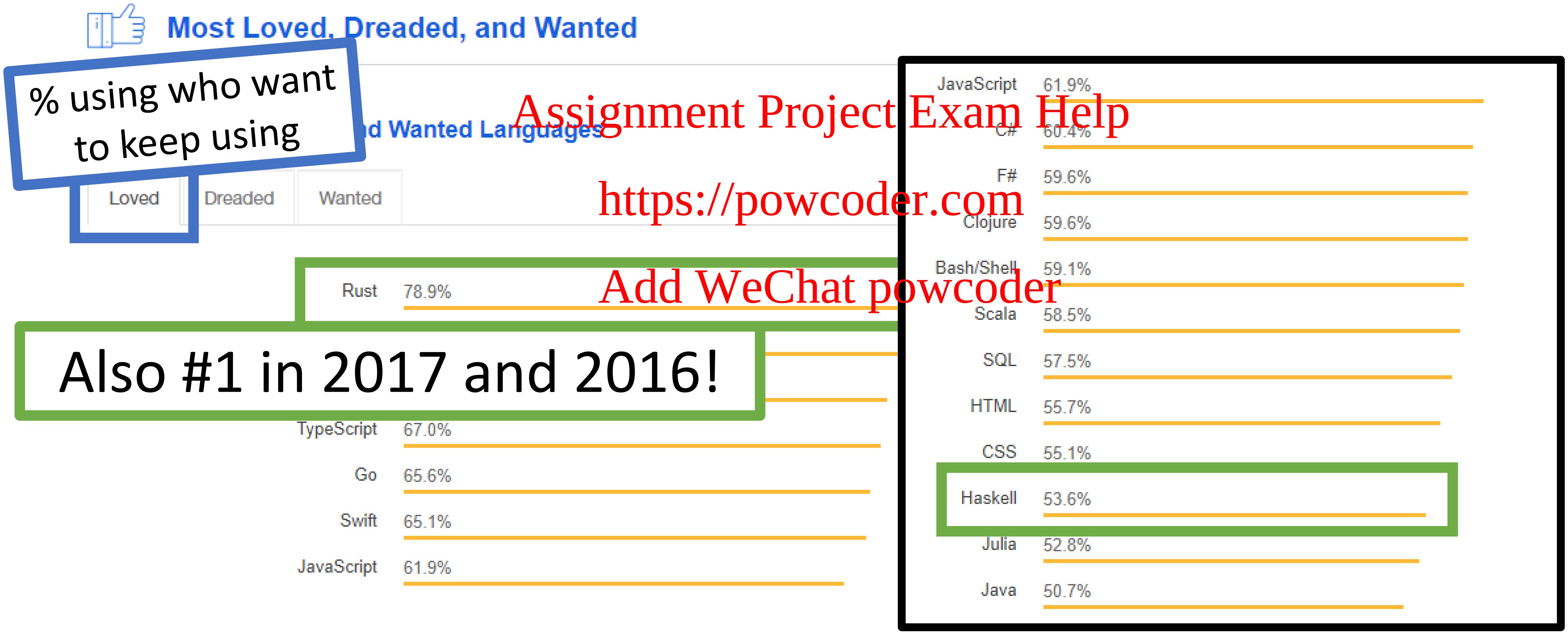
…to imperative.

Rust is an imperative language. However, we’ll see many cool features that remind us of the functional languages we’ve seen.

Rust History

- Grew out of a personal project by Mozilla employee Graydon Hoare in 2006
- Mozilla began sponsoring the project in 2009
- Officially announced in 2010
- Rust compiler successfully tested in 2011
- Pre-alpha version released in 2012
- Rust 1.0, the first stable release, arrived on May 15, 2015
- Youngest language we've seen so far
- Open source

IEEE Developer's Survey 2018



Most Loved, Dreaded, and Wanted

d Wanted Languages

Loved Dreaded Wanted

% using who want to keep using

Also #1 in 2017 and 2016!

Rust	78.9%
TypeScript	67.0%
Go	65.6%
Swift	65.1%
JavaScript	61.9%

JavaScript	61.9%
C#	60.4%
F#	59.6%
Clojure	59.6%
Bash/Shell	59.1%
Scala	58.5%
SQL	57.5%
HTML	55.7%
CSS	55.1%
Haskell	53.6%
Julia	52.8%
Java	50.7%

In Industry?

Mozilla in collaboration with Samsung

- Parallel web browser engine

Dropbox

- Magic Pocket file system, petabyte storage machines

Tor (The Onion Router)

- Experimenting with porting to Rust (from C) for safety features.

Rust Features

<u>**Systems Programming Language**</u>**:**

- In contrast with *application* programming languages.
- System software includes things like operating systems, utility software, device drivers, compilers, linkers, etc.
- System languages tend to feature more direct access to physical hardware of a given machine.

Rust Features

Syntax:

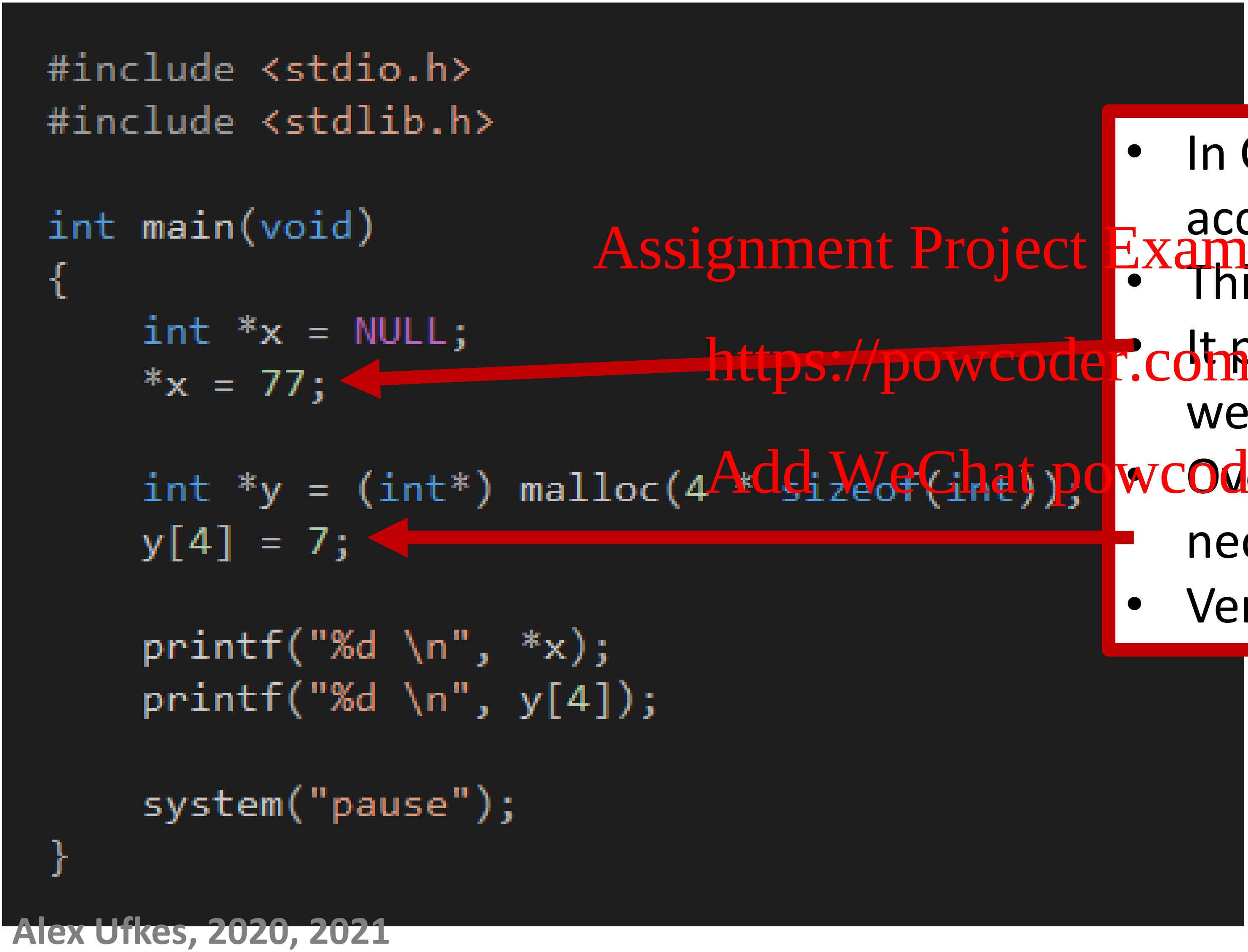
- Similar to C/C++
- Blocks of code delimited by { }
- Familiar control structures supported (`if`, `else`, `while`, `for`, etc.)
- Supports pattern matching! (`match`)
- Need not use `return`, last expression creates return value
- Functions largely composed of expressions

Rust Features

Memory Safety:

- Rust is designed to be *memory safe*
- Null or dangling pointers are not permitted.

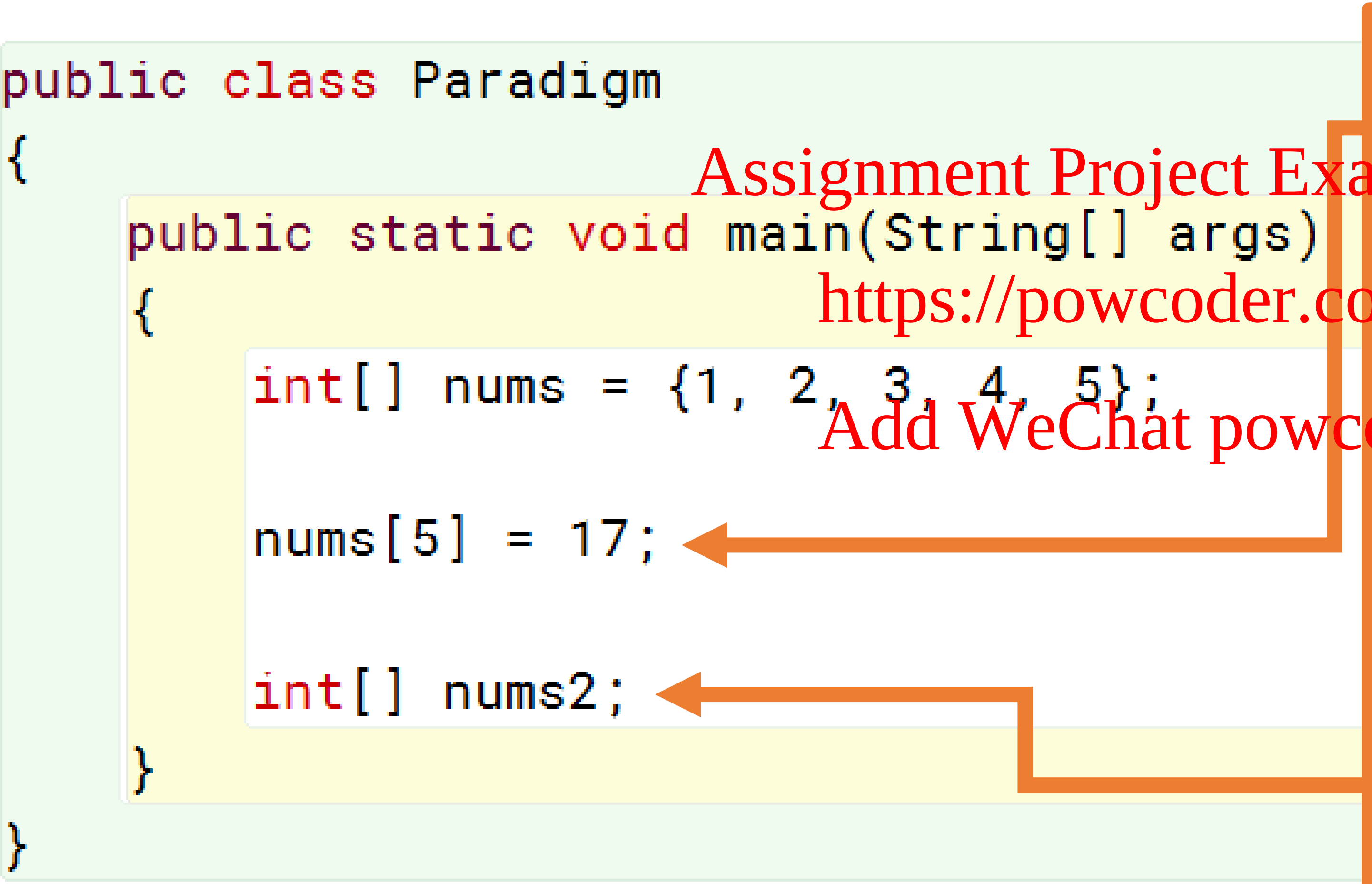
“Null or dangling pointers are not permitted”



```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = NULL;
    *x = 77;

    int *y = (int*) malloc(4 * sizeof(int));
    y[4] = 7;

    printf("%d \n", *x);
    printf("%d \n", y[4]);

    system("pause");
}
```

- In C, we’re allowed to ***<u>try</u>*** and access any memory we want.
- This code compiles!
- It produces a run-time error when we try and index into pointer x.
- Overrunning array bounds does not necessarily give a run time error!
- Very unsafe use of memory.

“Null or dangling pointers are not permitted”



```
public class Paradigm
{
    public static void main(String[] args)
    {
        int[] nums = {1, 2, 3, 4, 5};

        nums[5] = 17;

        int[] nums2;
    }
}
```

- Java is safer.
- This code *compiles*, but **always** throws an exception when we access outside array bounds.
- C/C++ only errors if going out of bounds accesses memory that your program doesn’t have write permission for.
- Java still allows dangling references.
- nums2 can be created without instantiating its object.

Rust Features

Memory Safety:

- Rust is designed to be *memory safe*
- Null or dangling pointers are not permitted.
- What about linked lists? Null pointers are useful.
- Rust defines an *option* type, which can be used to test if a pointer has *Some* value or *None*
  - What does this remind you of?

Rust Features

<u>**Memory Management**</u>**:**

- Rust does not do garbage collection
- ***Resource acquisition is initialization***
- RAII - Originated in C++
- Constructor used to acquire and initialize objects
- Resource *deallocation* is done by the destructor.
- No valid reference to object == no object.
- Not so in Java! Up to garbage collector.

Rust Features

Types and Polymorphism:

- Type system supports mechanism called "traits"
- Directly inspired by Haskell's type classes
- Supports type inference for variables declared with `let` keyword.
- Compile error if inference fails.
- Keyword `mut` for mutable variables.

Rust Features

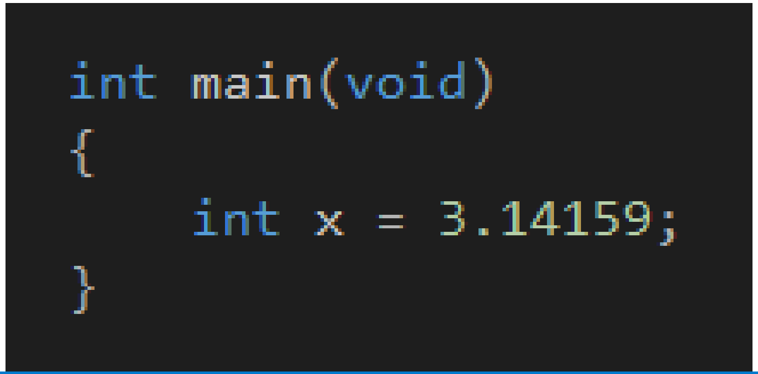
Pattern Matching:

- Rust supports pattern matching!
- Pattern matching is considered a sticking point for people learning Rust.
- We already have experience with it

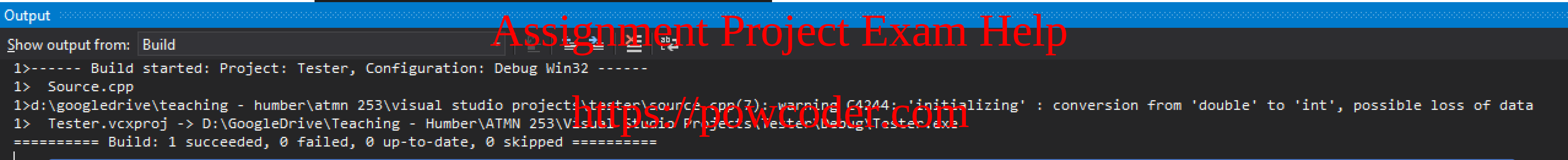
Rust & Safety

Strongly, statically typed

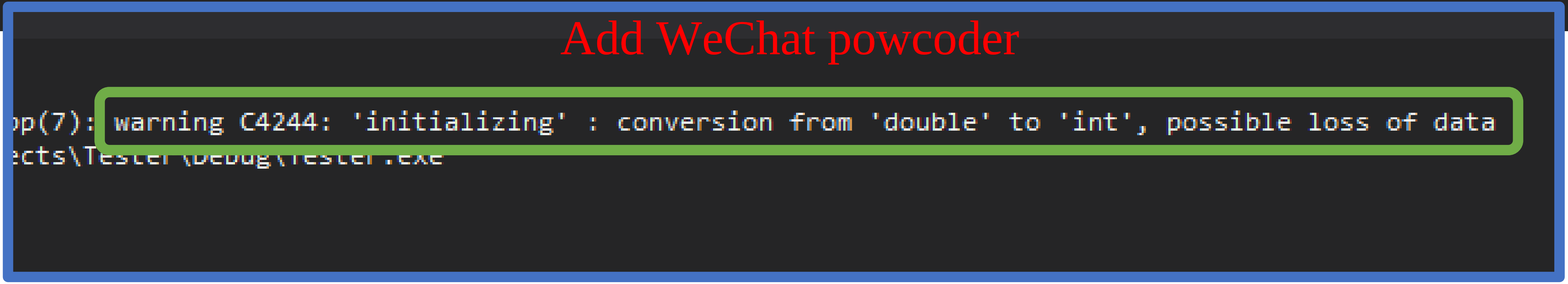
- Strong typing means limited implicit type conversions at compile time.
- C is happy to convert between numeric types without issue. Perhaps a compile warning in C++.
- Java raises compile error if there’s a loss of precision (double to float for example).



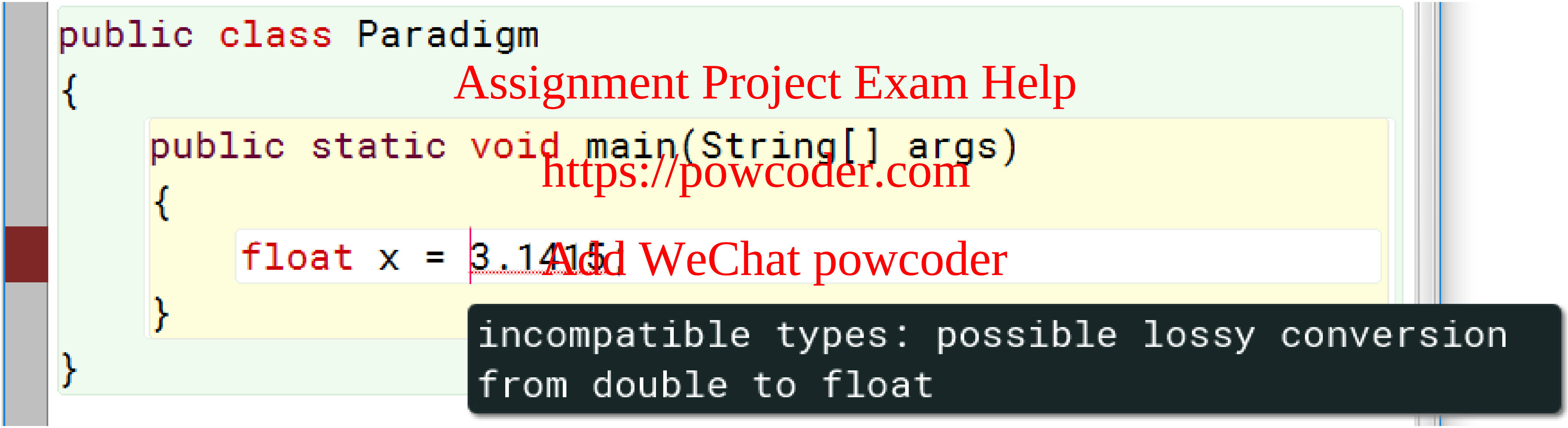
```
int main(void)
{
    int x = 3.14159;
}
```



```
1>------ Build started: Project: Tester, Configuration: Debug Win32 ------
1>  Source.cpp
1>d:\googledrive\teaching - humber\atmn 253\visual studio projects\tester\source.cpp(7): warning C4244: 'initializing' : conversion from 'double' to 'int', possible loss of data
1>  Tester.vcxproj -> D:\GoogleDrive\Teaching - Humber\ATMN 253\Visual Studio Projects\Tester\Debug\Tester.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```



```
pp(7): warning C4244: 'initializing' : conversion from 'double' to 'int', possible loss of data
ects\Tester\Debug\Tester.exe
```



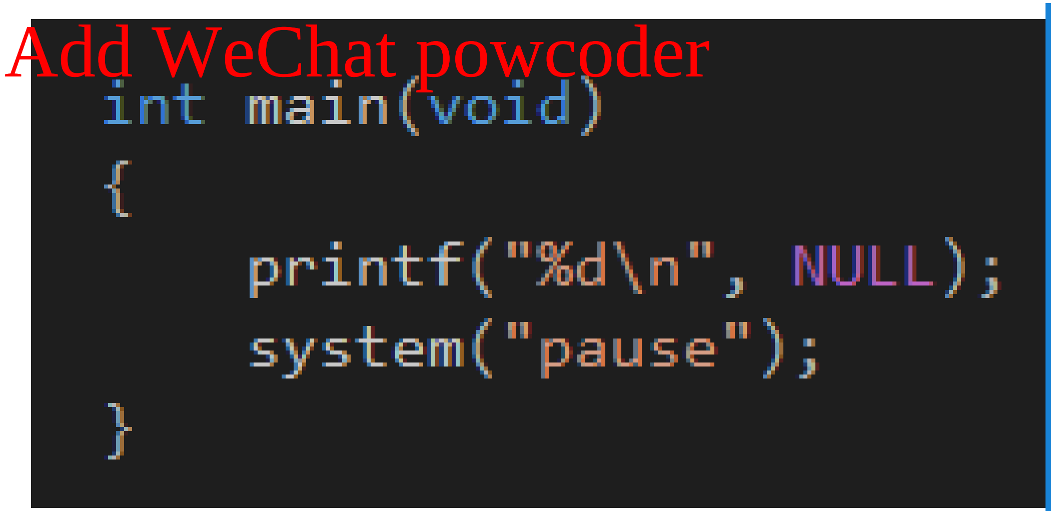
```
public class Paradigm
{
    public static void main(String[] args)
    {
        float x = 3.1415
    }
}
```

```
incompatible types: possible lossy conversion
from double to float
```

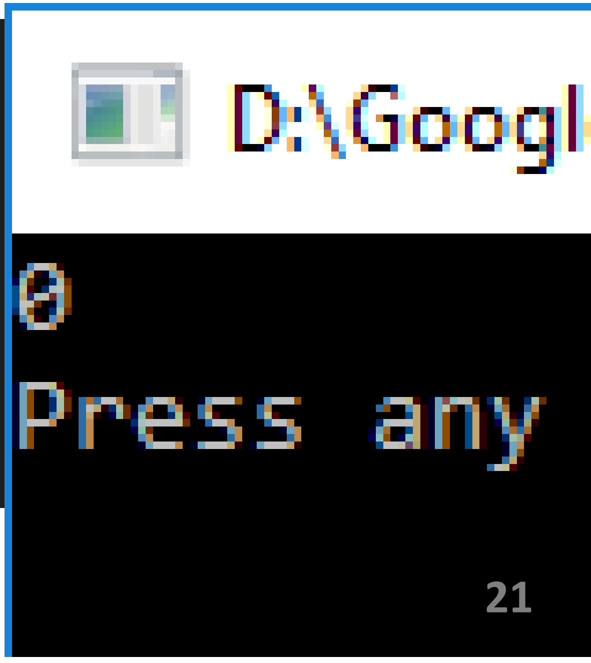
Rust & Safety

No “Undefined Behavior”

- Null pointer dereferencing
    - Attempt to dereference address 0



```
int main(void)
{
    printf("%d\n", NULL);
    system("pause");
}
```

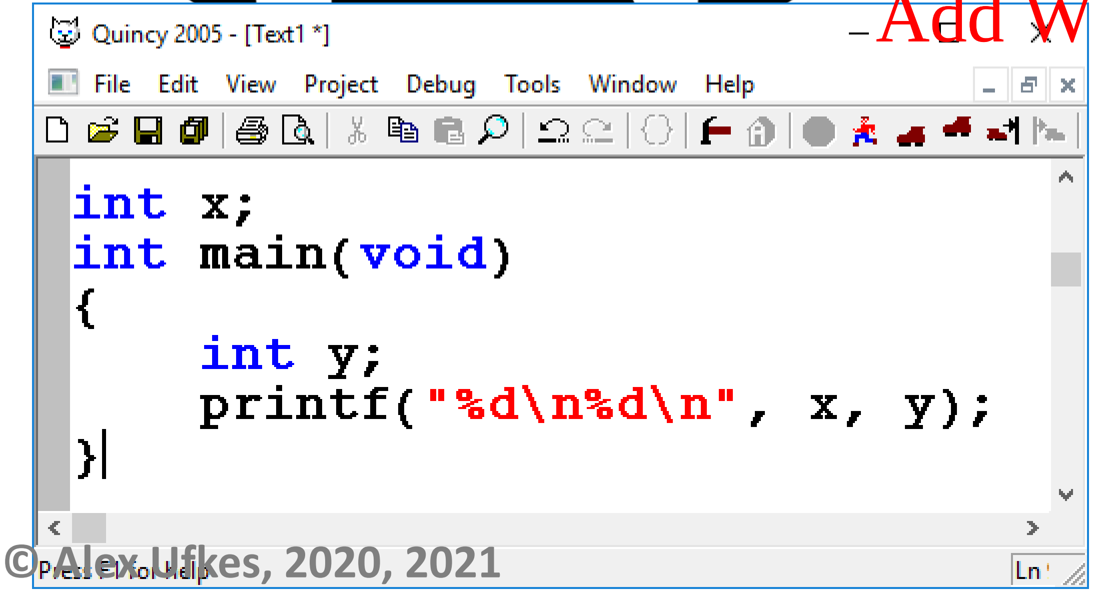


```
0
Press any
```

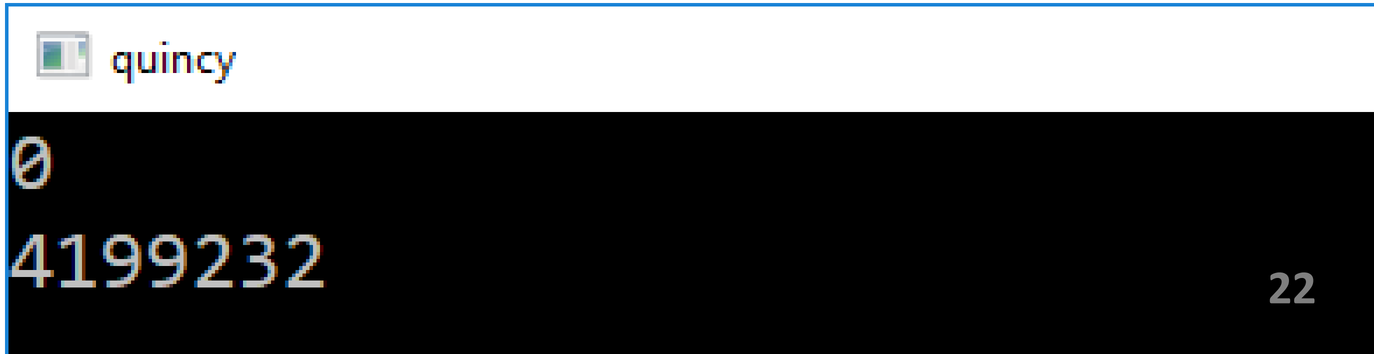
Rust & Safety

No “Undefined Behavior”

- Null pointer dereferencing
  - ○ Attempt to dereference address 0
- Use of variable before it’s initialized
  - ○ In C, we get whatever was in memory before that.
  - ○ Only globals auto-initialize to 0



```c
int x;
int main(void)
{
    int y;
    printf("%d\n%d\n", x, y);
}
```

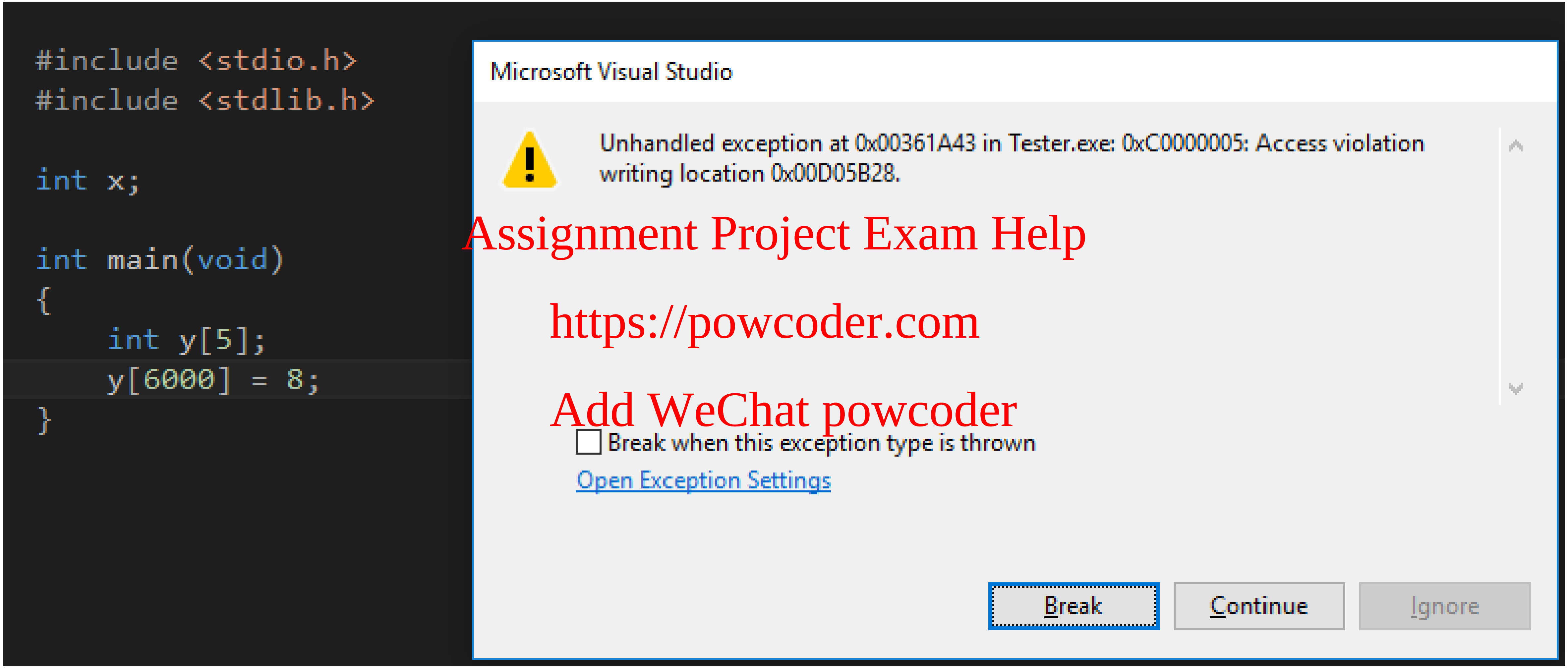


```
0
4199232
```

Rust & Safety

No "Undefined Behavior"

- Null pointer dereferencing
  - Attempt to dereference address 0
- Use of variable before it's initialized
  - In C, we get whatever was in memory before that.
  - Only globals auto-initialize to 0
- Array index out of bounds
  - May or may not cause runtime error (in C), depends who owns memory



```
#include <stdio.h>
#include <stdlib.h>

int x;

int main(void)
{
    int y[5];
    y[6000] = 8;
}
```

Microsoft Visual Studio

Unhandled exception at 0x00361A43 in Tester.exe: 0xC0000005: Access violation writing location 0x00D05B28.

□ Break when this exception type is thrown

Open Exception Settings

Rust & Safety

No "Undefined Behavior"

- Signed integer overflow & optimization

X+1 > X

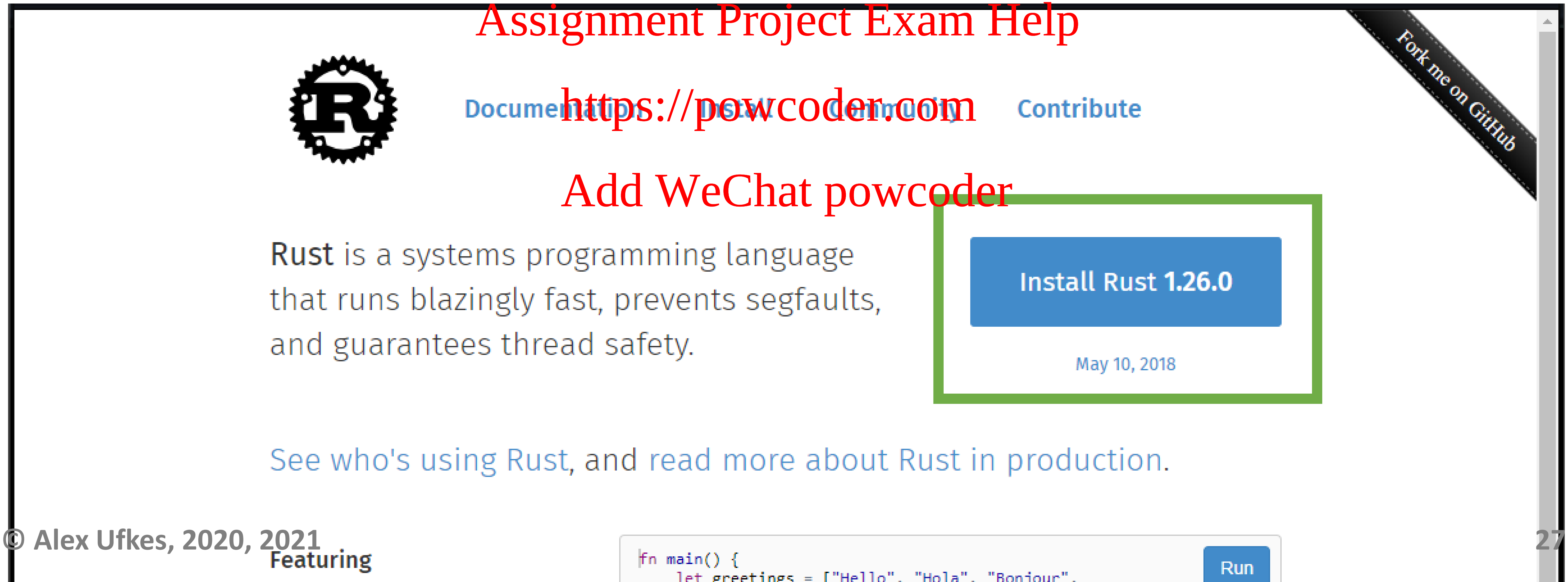
- If overflow is undefined, compiler can just optimize this to simply **true**.
- Dangerous if X can overflow!
- Forcing compiler to consider overflow means we lose certain optimizations.

Rust Non-Goals

- We do not employ any particularly cutting-edge technologies. Old, established techniques are better.
- We do not prize expressiveness, minimalism or elegance above other goals. These are desirable but subordinate goals.
- We do not intend to cover the complete feature-set of C++, or any other language. Rust should provide majority-case features.
- We do not intend to be 100% static, 100% safe, 100% reflective, or too dogmatic in any other sense. Trade-offs exist.
- We do not demand that Rust run on “every possible platform”. It must eventually work without unnecessary compromises on widely-used hardware and software platforms.

Installing Rust

https://www.rust-lang.org/en-US/index.html



Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

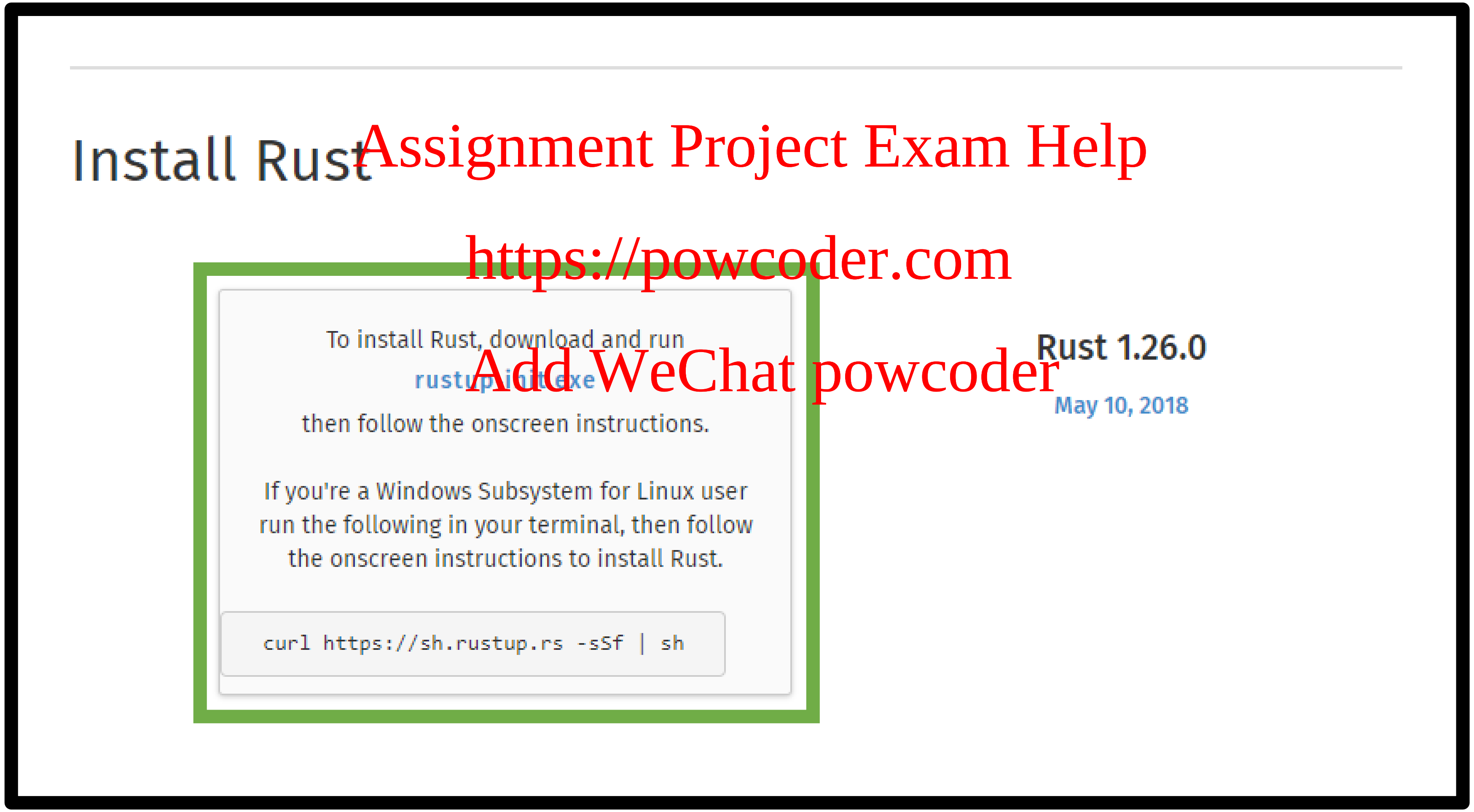
May 10, 2018

See who's using Rust, and read more about Rust in production.

Featuring

```
fn main() {
    let greetings = ["Hello", "Hola", "Bonjour",
```

Installing Rust



Install Rust

To install Rust, download and run
rustup-init.exe
then follow the onscreen instructions.

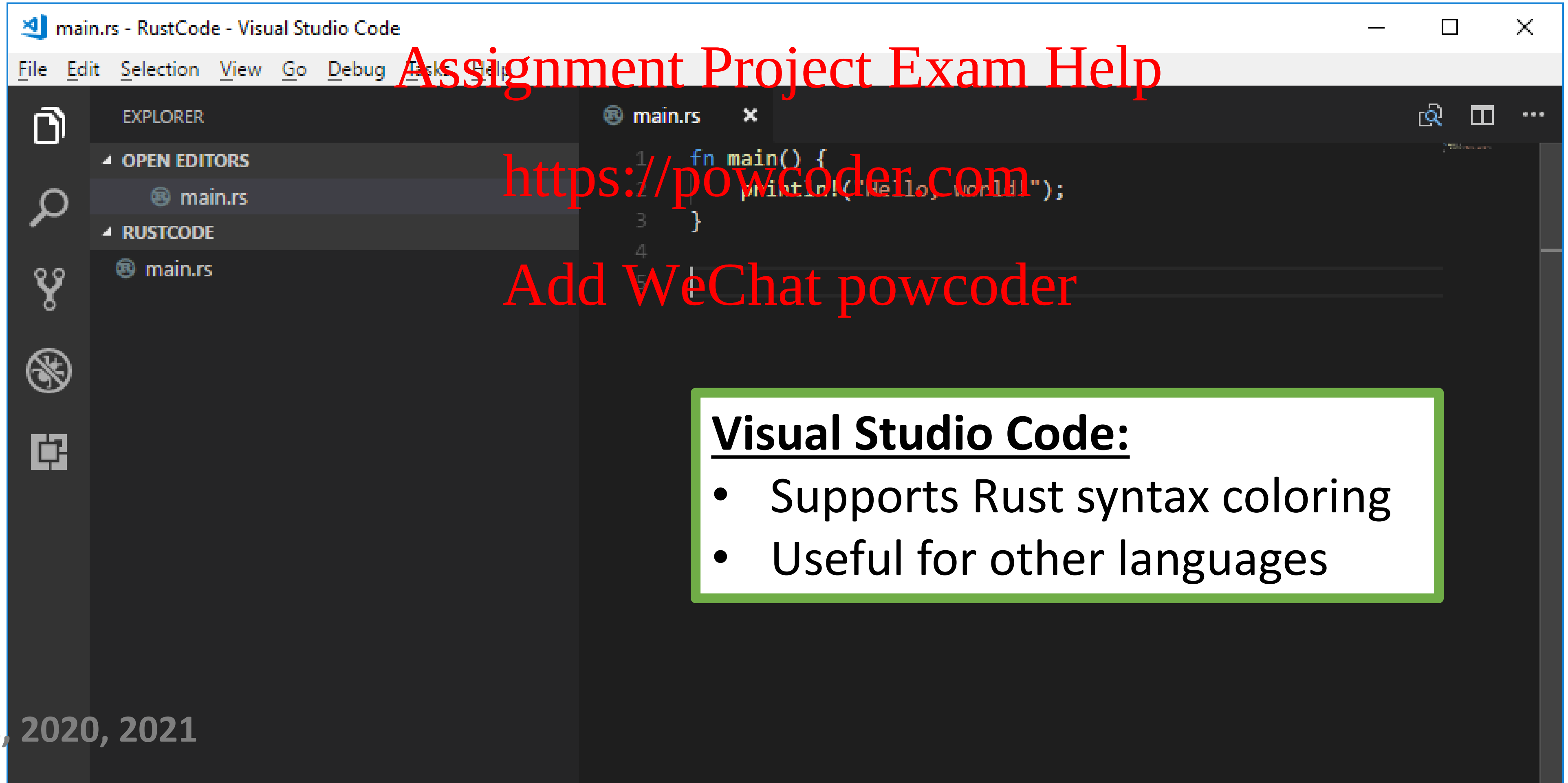
If you're a Windows Subsystem for Linux user run the following in your terminal, then follow the onscreen instructions to install Rust.

```
curl https://sh.rustup.rs -sSf | sh
```

Rust 1.26.0
May 10, 2018

Editing Rust Code

Any text editor will do, but I like VSCode:



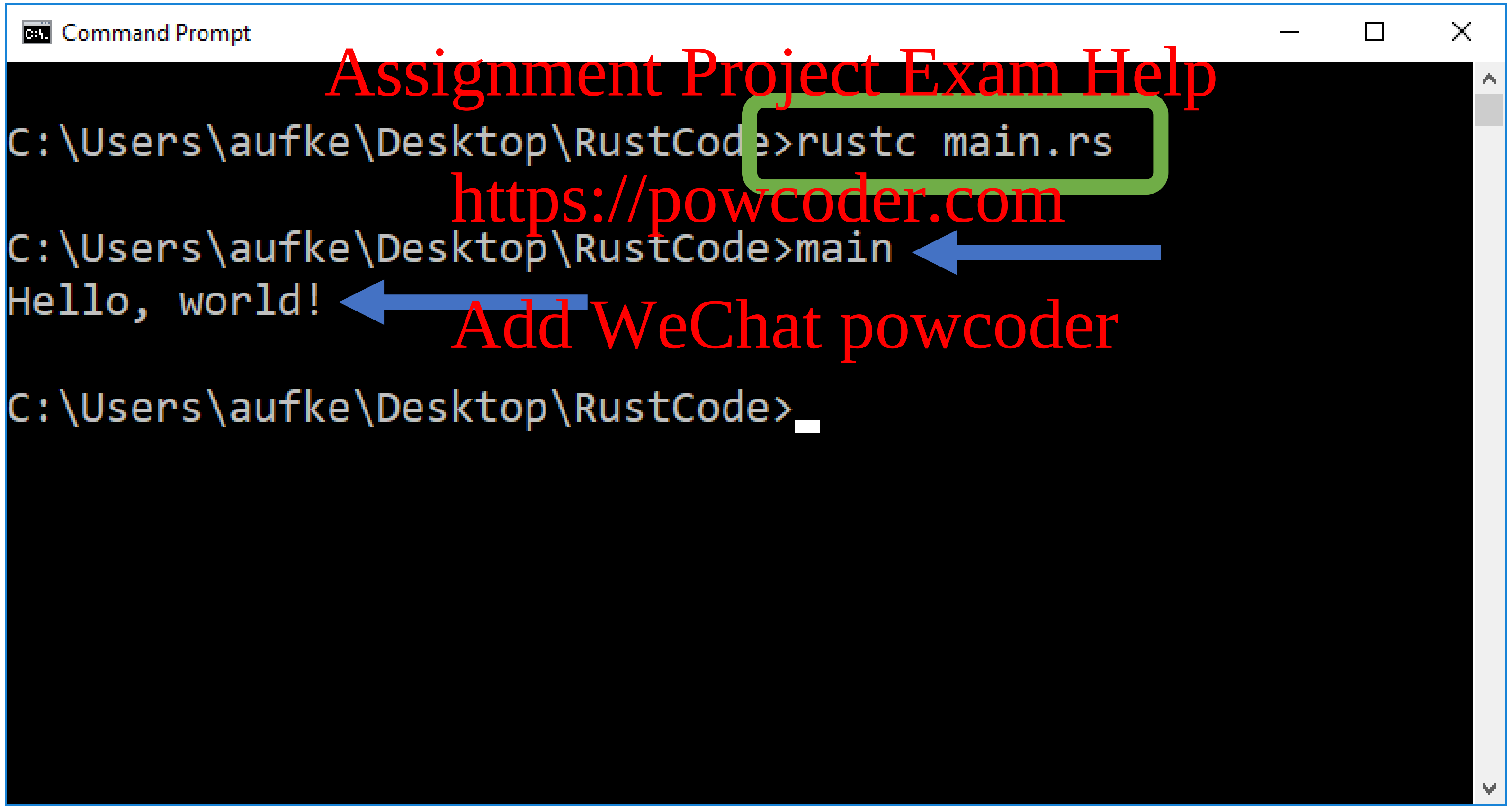
```
fn main() {
    println!("Hello, world!");
}
```

Visual Studio Code:

- Supports Rust syntax coloring
- Useful for other languages

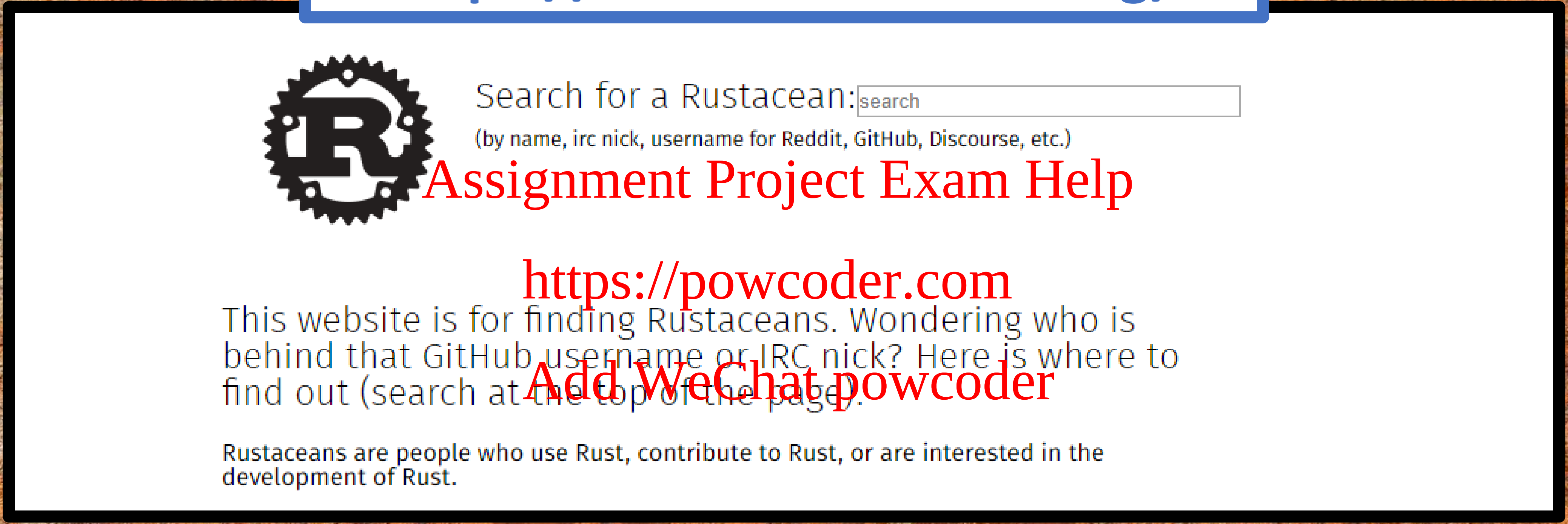
Compiling Rust Code

Command Line - **rustc**



```
C:\Users\aufke\Desktop\RustCode>rustc main.rs

C:\Users\aufke\Desktop\RustCode>main
Hello, world!

C:\Users\aufke\Desktop\RustCode>_
```

https://www.rustaceans.org/



Search for a Rustacean: search

(by name, irc nick, username for Reddit, GitHub, Discourse, etc.)

This website is for finding Rustaceans. Wondering who is behind that GitHub username or IRC nick? Here is where to find out (search at the top of the page).

Rustaceans are people who use Rust, contribute to Rust, or are interested in the development of Rust.

HOT ~~C~~RUSTACEANS
NEAR YOU LOOKING TO
SCUTTLE
Join now ▶
https://www.facebook.com/crabuardo/

Much of the syntax is reminiscent of C/C++

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

Like C, C++, Java, Haskell, and many others, main() defines the entry point for executing a Rust program.

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

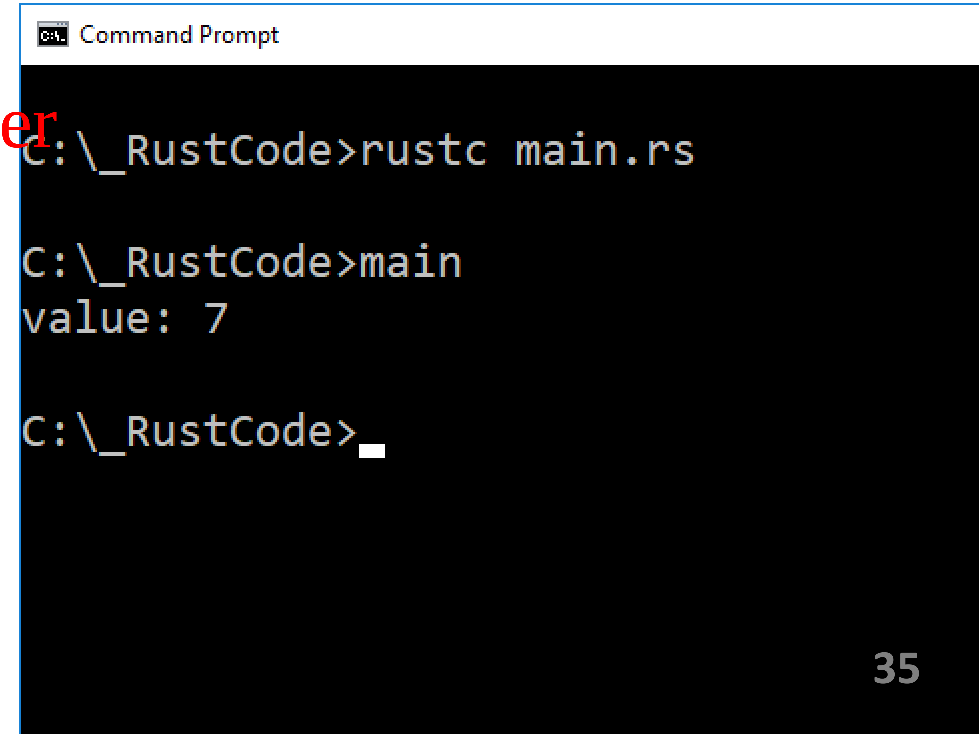
`println` vs `println!`

- The ! indicates we’re calling a macro.
- A standard function call doesn’t include !

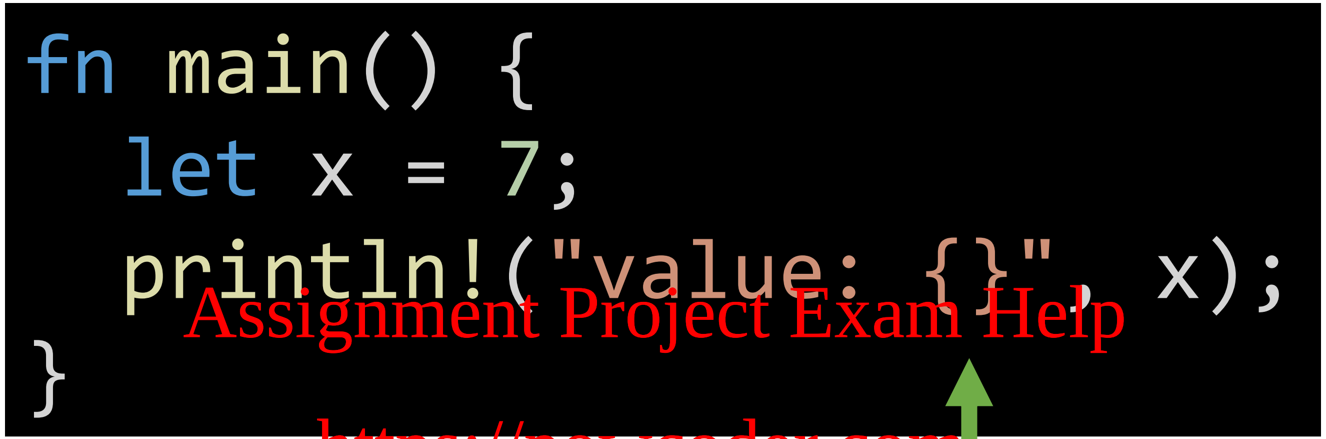
Variables

- By default, Rust variables are immutable
- Once initialized, can’t change.
- Like `final` or `const` in other languages
- Declare using `let` keyword:

```
fn main() {
  let x = 7;
  println!("value: {}", x);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
value: 7

C:\_RustCode>_
```

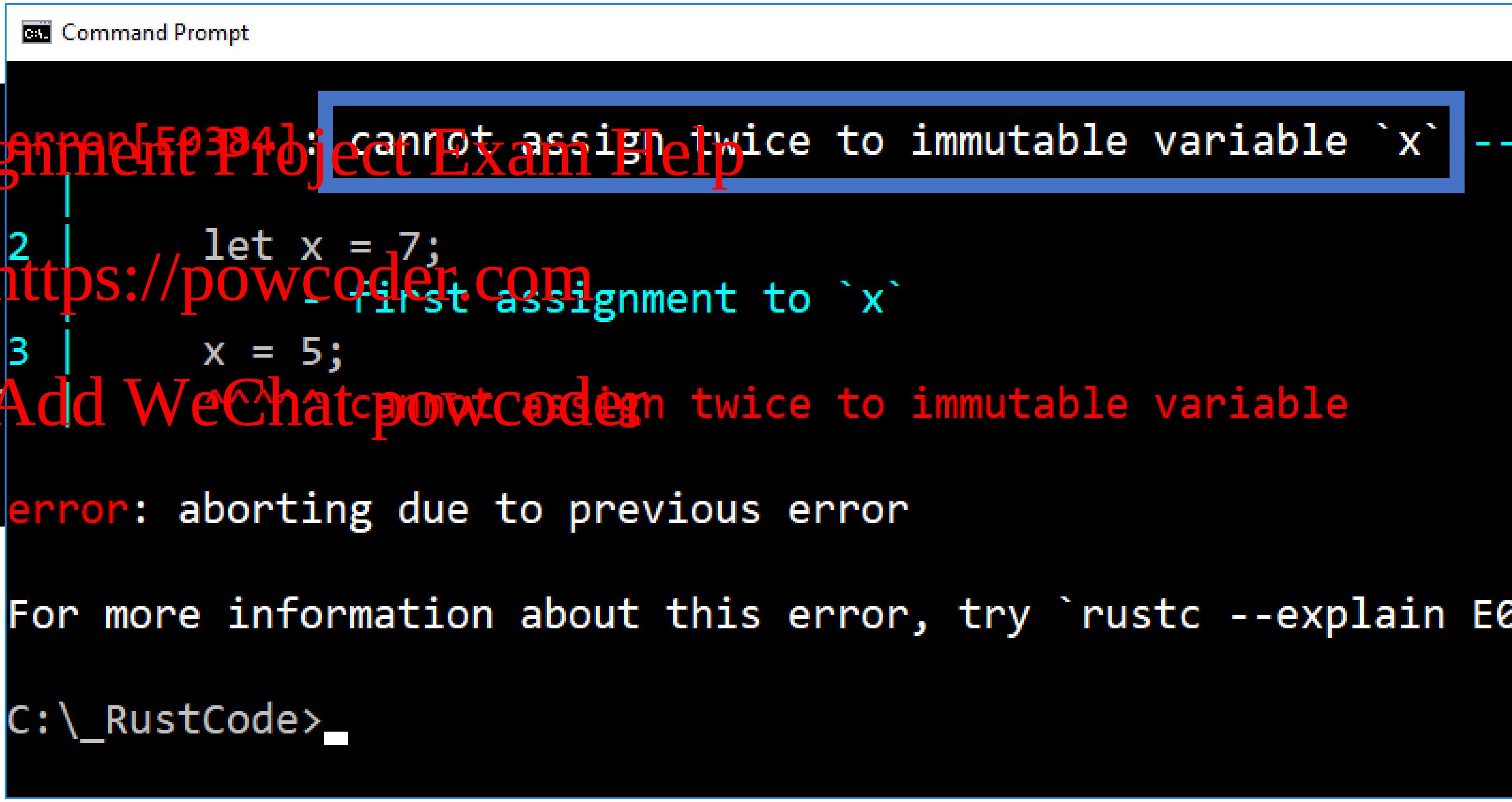


```rust
fn main() {
  let x = 7;
  println!("value: {}", x);
}
```

Curly brace pair in a `println` string acts as a C/C++ style placeholder

Variables

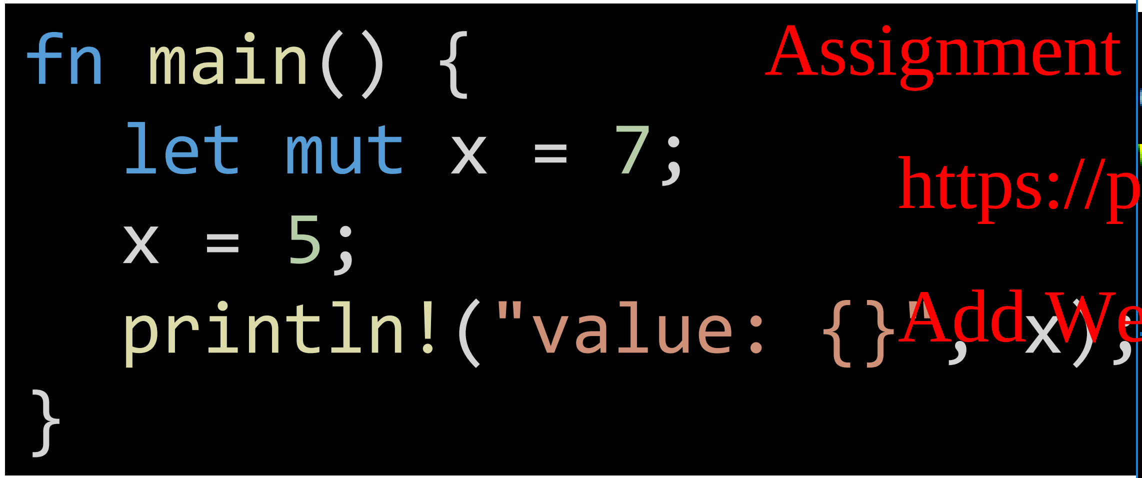
```
fn main() {
   let x = 7;
   x = 5;
   println!("value:
}
```



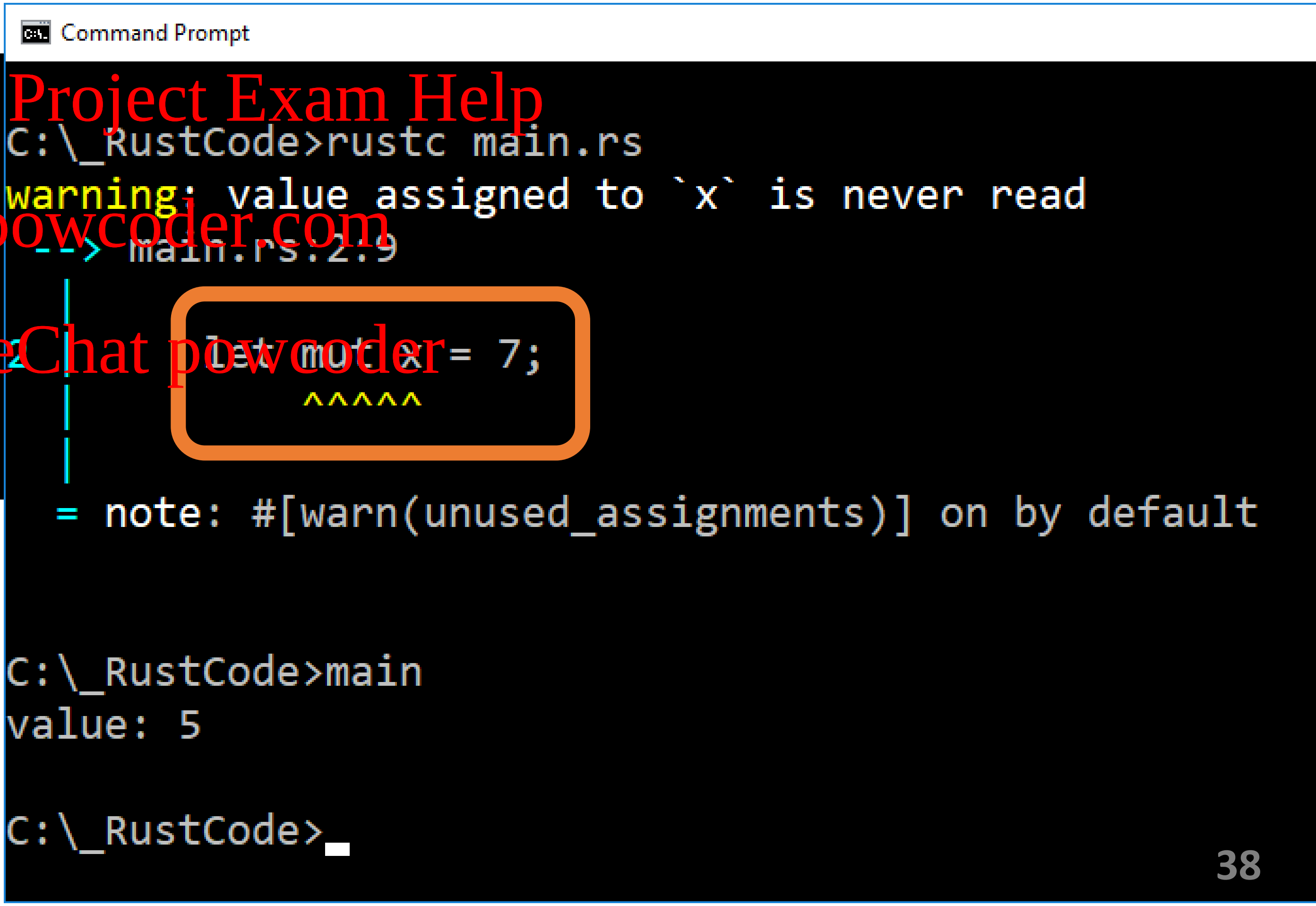
```
error[E0384]: cannot assign twice to immutable variable `x` --
  |
2 |     let x = 7;
  |         - first assignment to `x`
3 |     x = 5;
  |     ^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0

C:\_RustCode>_
```

Mutable Variables

Use `mut` keyword:



```
fn main() {
   let mut x = 7;
   x = 5;
   println!("value: {}", x);
}
```

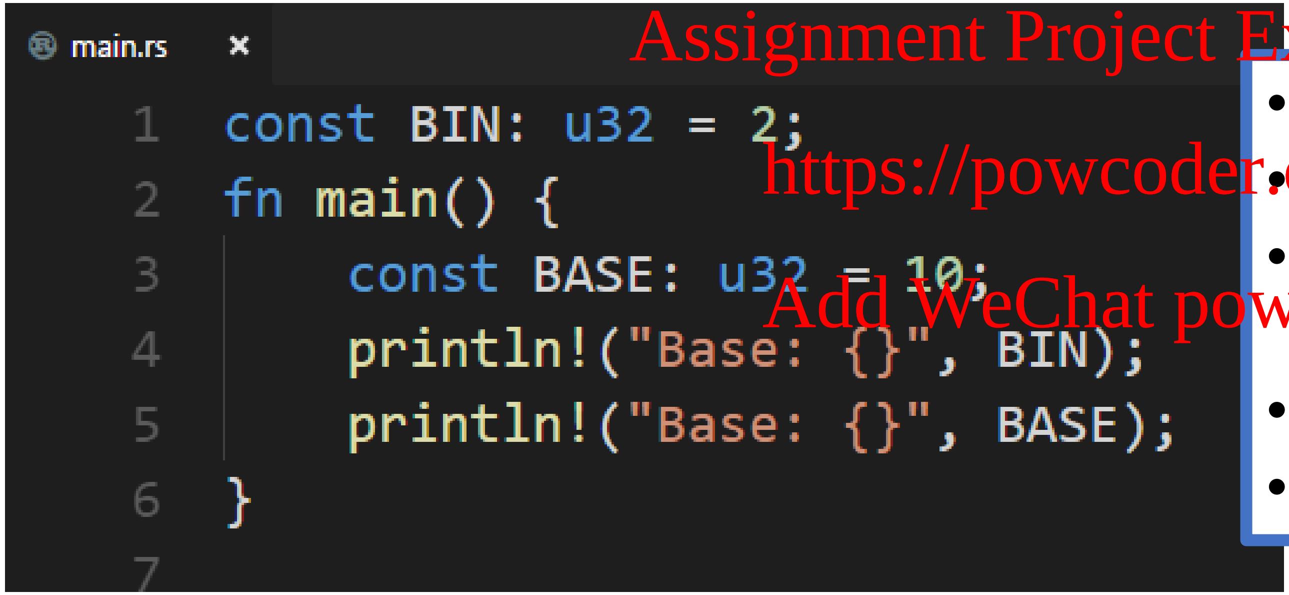


```
C:\_RustCode>rustc main.rs
warning: value assigned to `x` is never read
 --> main.rs:2:9
  |
2 |     let mut x = 7;
  |         ^^^^^
  |
  = note: #[warn(unused_assignments)] on by default


C:\_RustCode>main
value: 5

C:\_RustCode>
```

- We get a warning, and it's sensible.
- We change the value of x before the initial value is ever read.
- Pointless.

Constant/Global Variables

Rust still has them:

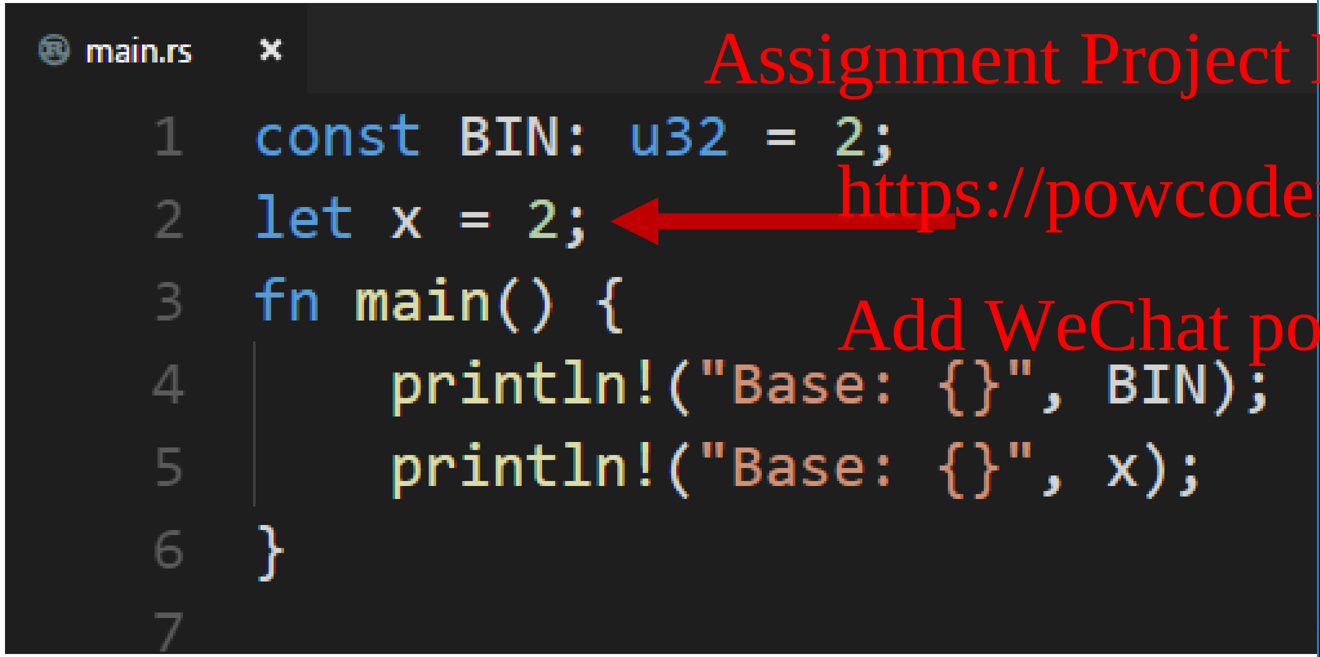


```rust
const BIN: u32 = 2;
fn main() {
    const BASE: u32 = 10;
    println!("Base: {}", BIN);
    println!("Base: {}", BASE);
}
```

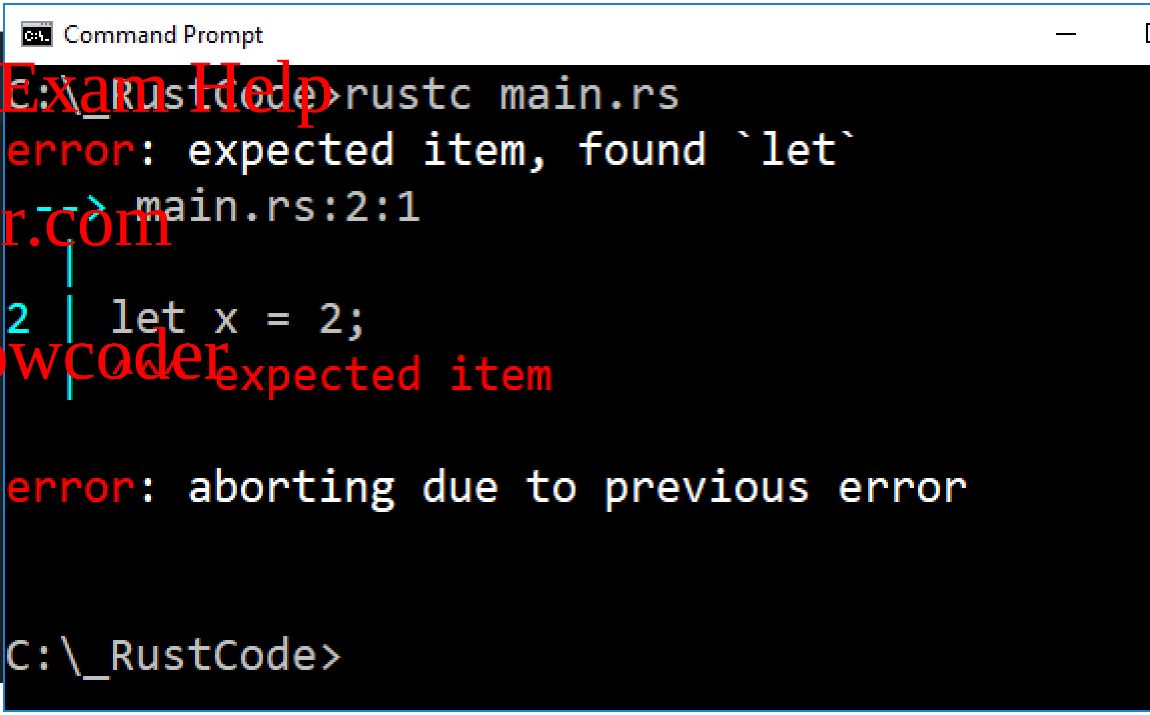
- Use `const` instead of `let`
- *Always* immutable
- Can be declared in global scope, unlike `let`
- Must indicate data type (u32)
- More on types coming up.

Constant/Global Variables

Can be declared in global scope, unlike **let**



```rust
const BIN: u32 = 2;
let x = 2;
fn main() {
    println!("Base: {}", BIN);
    println!("Base: {}", x);
}
```



```
C:\_RustCode>rustc main.rs
error: expected item, found `let`
 --> main.rs:2:1
  |
2 | let x = 2;
  | ^^^ expected item

error: aborting due to previous error


C:\_RustCode>
```

Shadowing

Variables with the same name?

In Java, variables can have the same name so long as their scope does not overlap:

BAD

```
int r = 10;
if (x >= 0) {
        double r = Math.sqrt(x);
}
```

OK

```
if (x >= 0) {
   double r = Math.sqrt(x); }
else {
   float r = 0; }
```

Shadowing

Variables with the same name?

C++ is less strict. Scopes can overlap, but they can't be identical:

```
int r = 10;
if (x >= 0) {
    double r = sqrt(4.0);
}
```

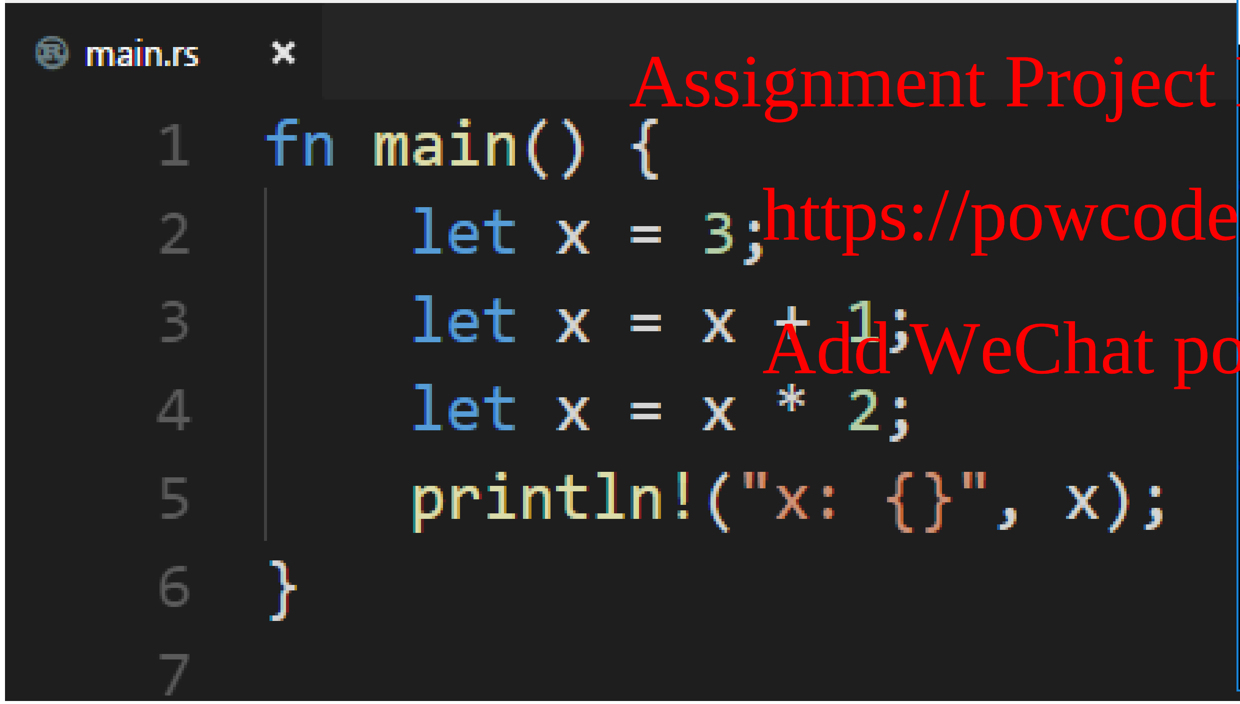
OK

```
if (x >= 0) {
    double r = sqrt(4.0);
    float r = 0;
}
```

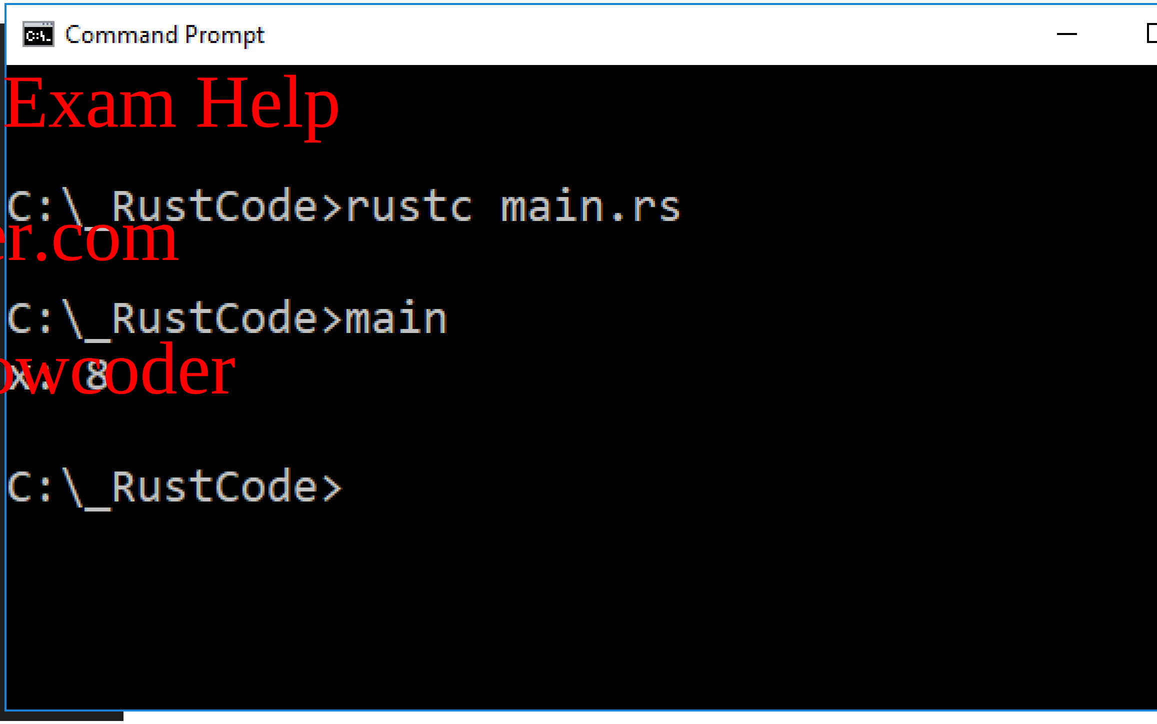
BAD

Shadowing

Variables with the same name?



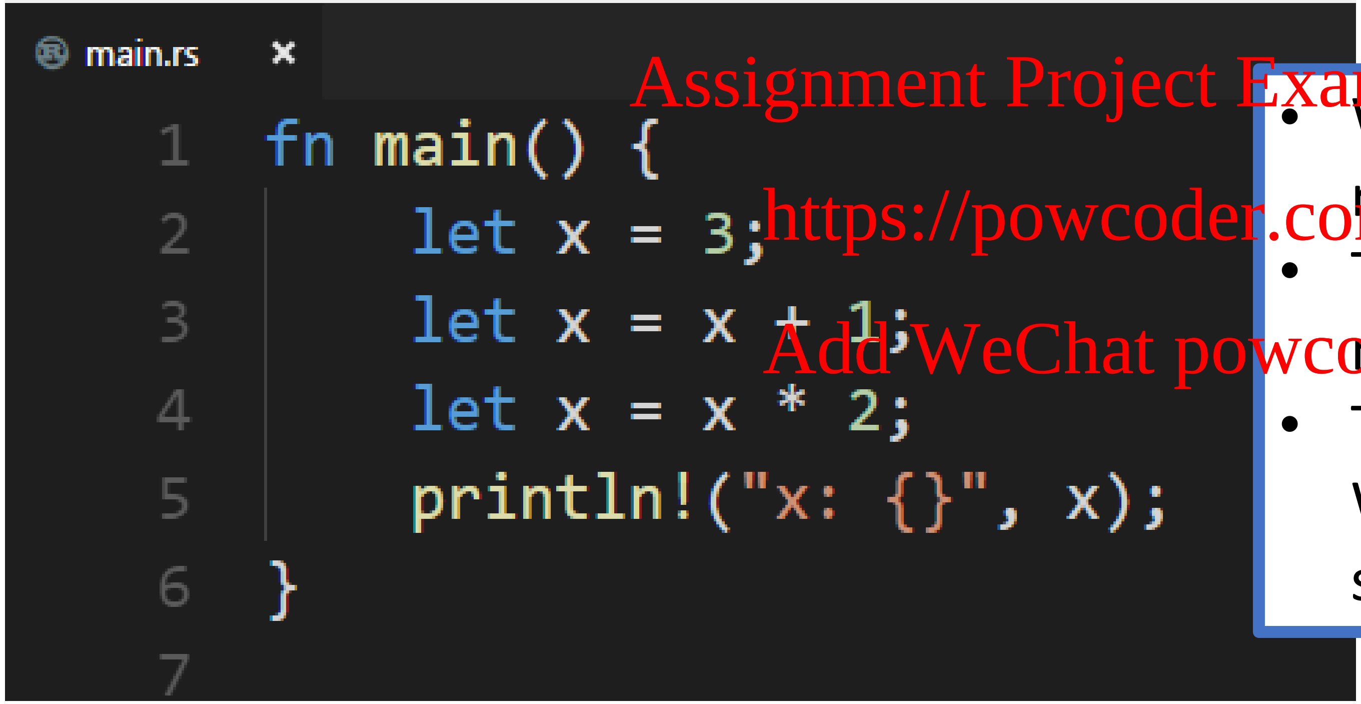
```
fn main() {
    let x = 3;
    let x = x + 1;
    let x = x * 2;
    println!("x: {}", x);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
x: 8

C:\_RustCode>
```

Shadowing

Variables with the same name?

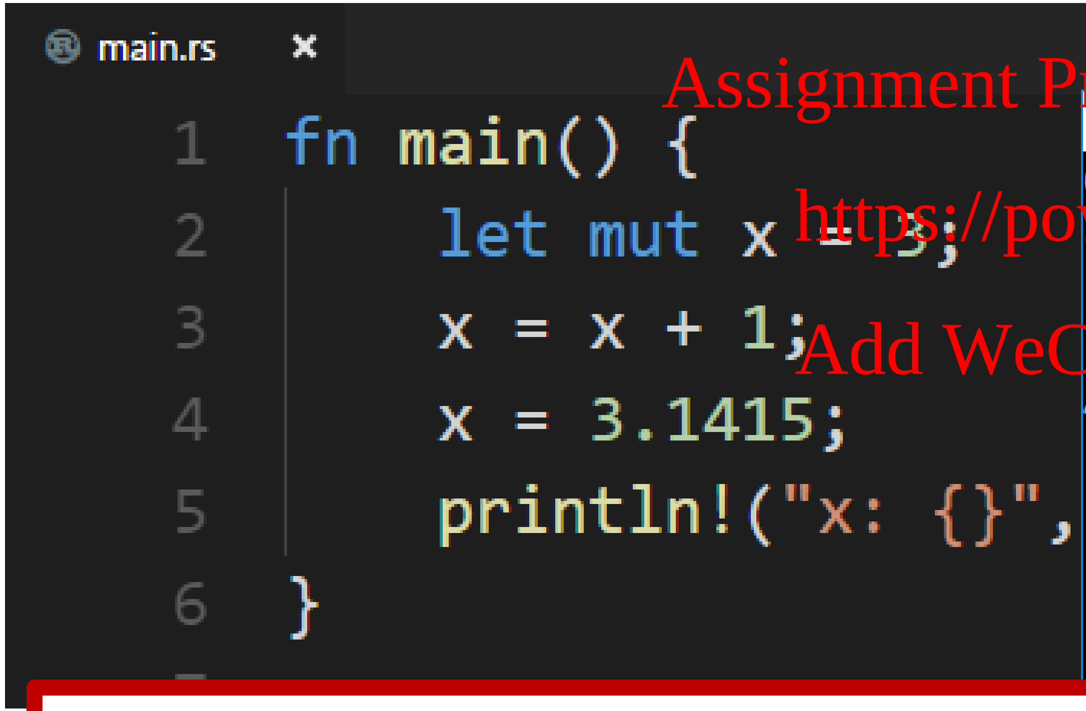


```
fn main() {
    let x = 3;
    let x = x + 1;
    let x = x * 2;
    println!("x: {}", x);
}

```

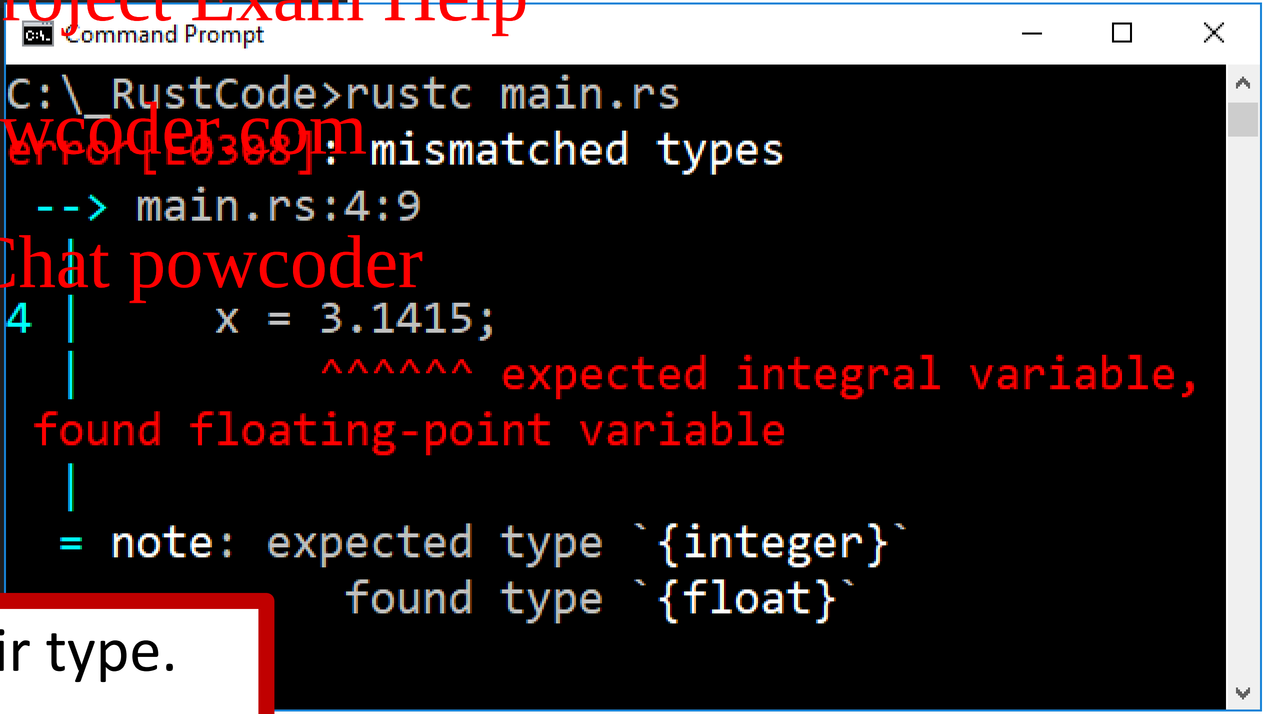
- What we're doing here is like re-binding in Haskell or Elixir.
- This doesn't work with mutable variables.
- Think of this mathematically – We're simply saying let x = something else.

Shadowing VS mut

Why not just use shadowing? Why do we need mut?



```
fn main() {
    let mut x = 5;
    x = x + 1;
    x = 3.1415;
    println!("x: {}",
}
```

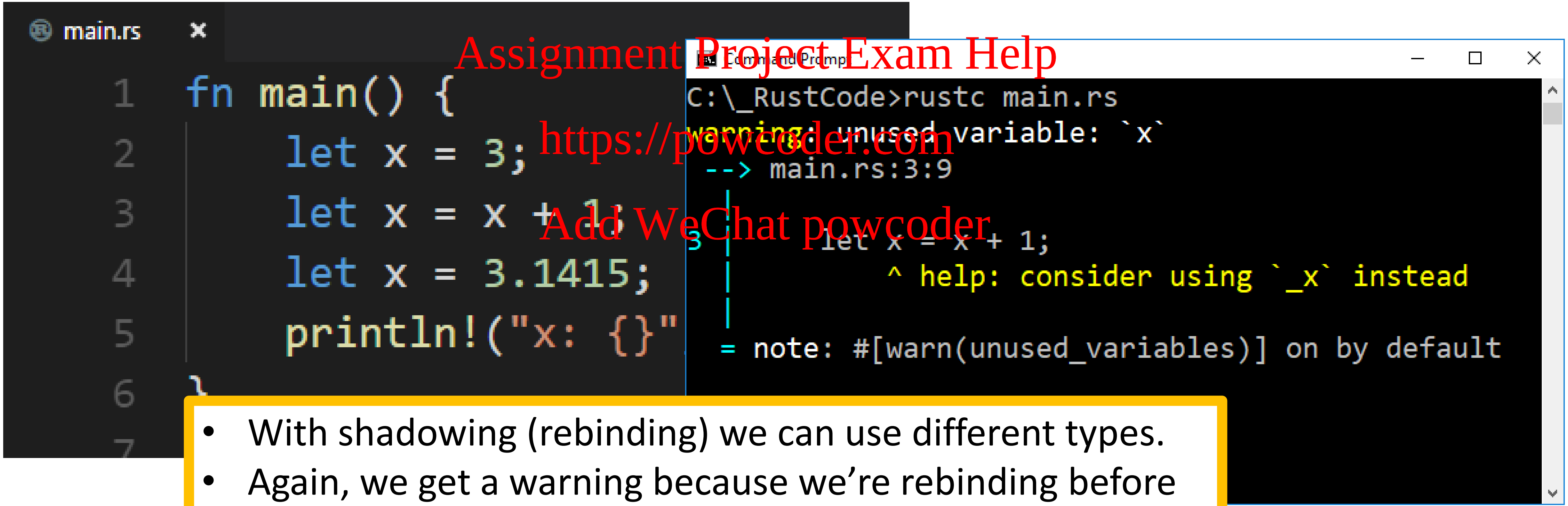


```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:4:9
  |
4 |     x = 3.1415;
  |         ^^^^^^ expected integral variable,
 found floating-point variable
  |
  = note: expected type `{integer}`
             found type `{float}`
```

- Mutable variables are stuck with their type.
- Can't assign a value of a different type.

Shadowing VS mut

Why not just use shadowing? Why do we need mut?



```
fn main() {
    let x = 3;
    let x = x + 1;
    let x = 3.1415;
    println!("x: {}"
}
```

```
C:\_RustCode>rustc main.rs
warning: unused variable: `x`
 --> main.rs:3:9
  |
3 |     let x = x + 1;
  |         ^ help: consider using `_x` instead
  |
  = note: #[warn(unused_variables)] on by default
```

- With shadowing (rebinding) we can use different types.
- Again, we get a warning because we're rebinding before the original binding is ever used.

Shadowing VS `mut`

Why not just use shadowing? Why do we need `mut`?

- With `mut`, we're *mutating* a variable in memory.
- Storing a different value in the same variable.
- The name still refers to the same place, thus the **type must stay the same**.

- With shadowing, we're getting a new variable in memory each time.
- We're changing what a given name is referring to.
- We're not changing the existing value.

Data Types

Two subsets: Scalar and Compound

Reminder: Rust is statically typed. Must know all variable types at compile time.

Scalar types represent a single value:

- Rust has four: integers, floating-point, Booleans, characters.

Compound types group multiple values:

- Two primitive compound types: tuples and arrays.

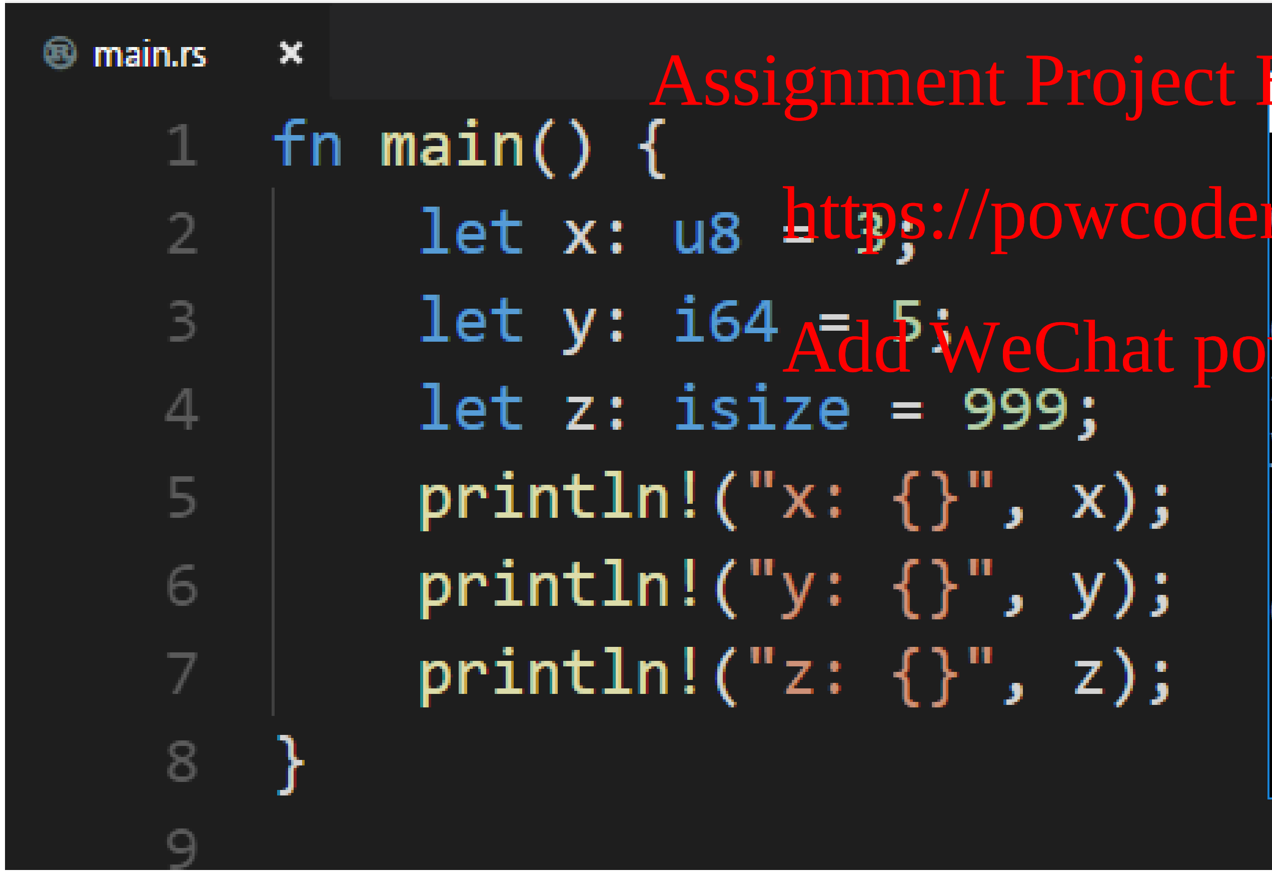
Scalar Types: Integers

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

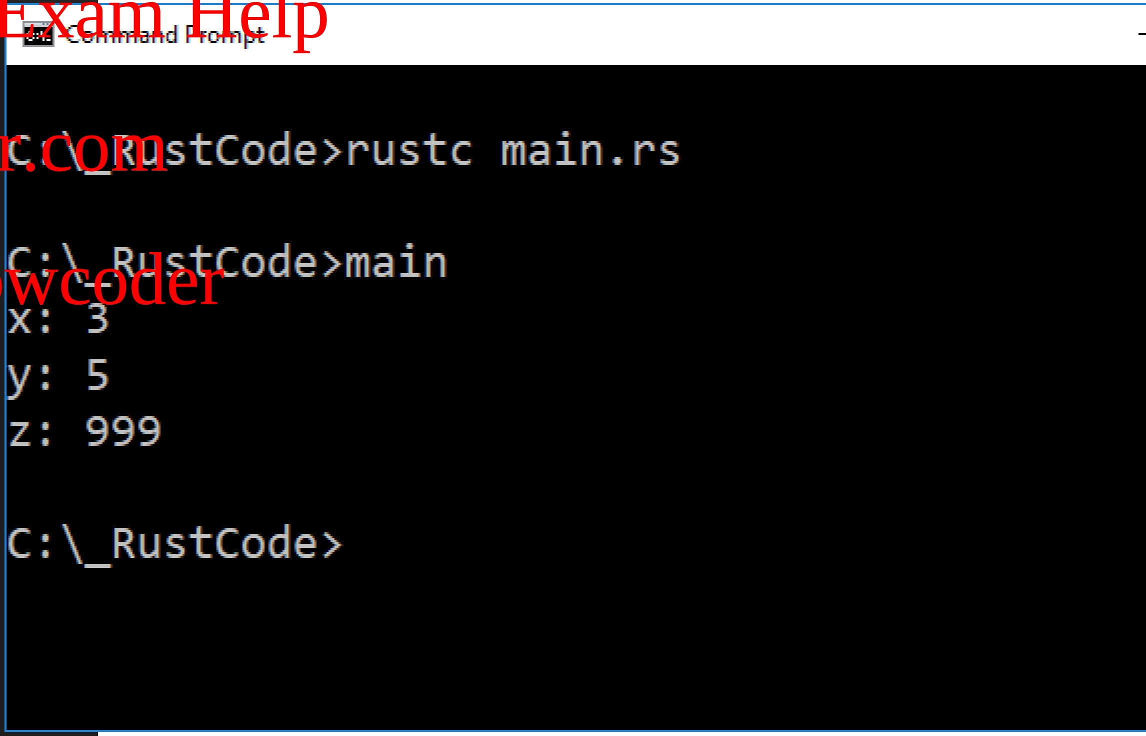
- Signed integers are stored using 2s comp
- Arch will be 32 bits on a 32 bit system, 64 bits on a 64 bit system.
- When not specified, Rust defaults to i32

Specify Type?

Rust has type inference, but we can be explicit:



```rust
fn main() {
    let x: u8 = 3;
    let y: i64 = 5;
    let z: isize = 999;
    println!("x: {}", x);
    println!("y: {}", y);
    println!("z: {}", z);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
x: 3
y: 5
z: 999

C:\_RustCode>
```

Integer Literals

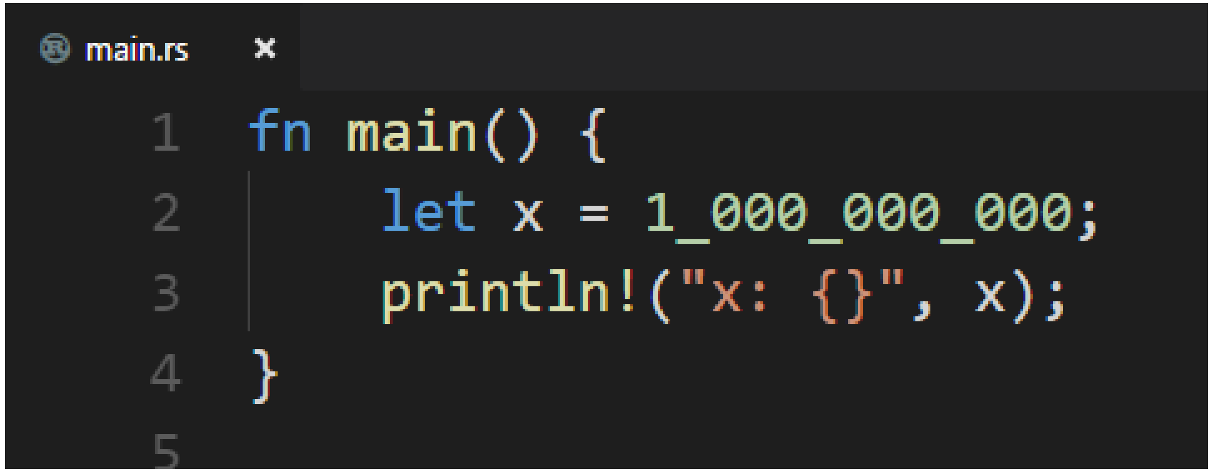
In addition to just writing the value…

Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

Bytes can be character literals

Notice the _

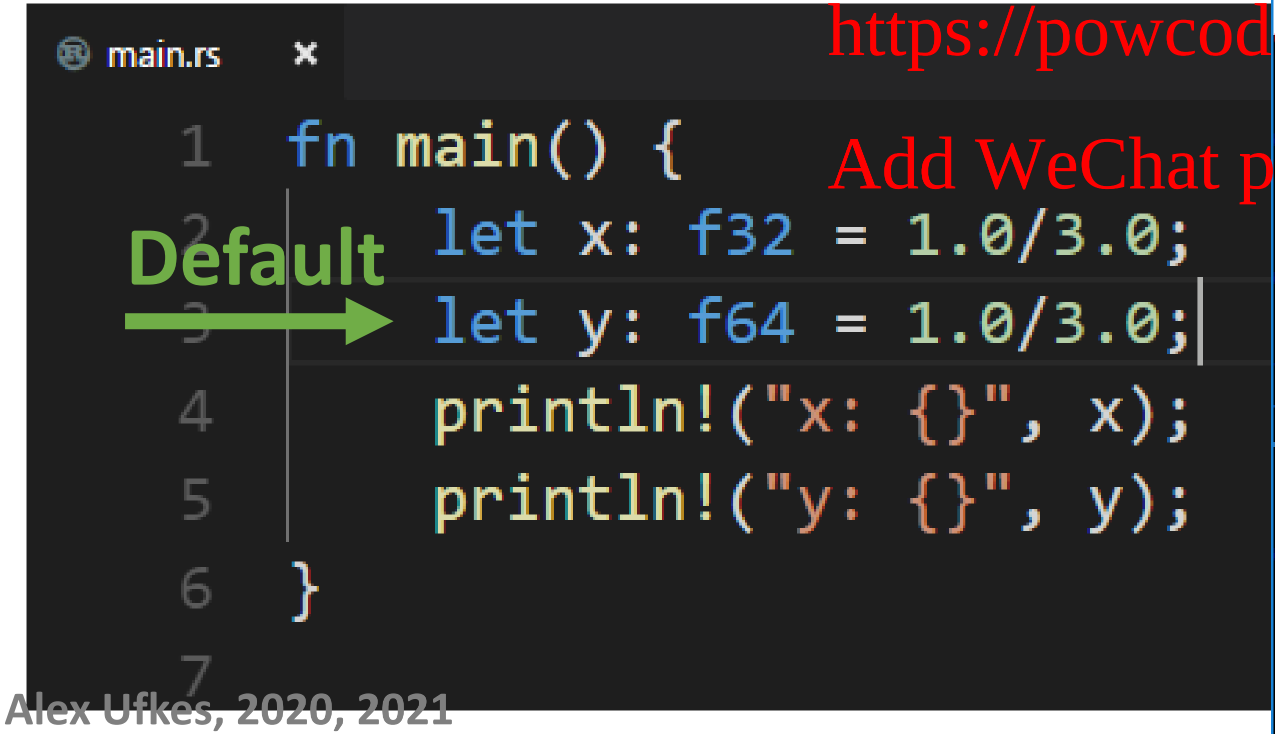
- This is a handy visual sugar
- Hard to count the zeroes in 1000000000. What number is this?
- Easy to see 1_000_000_000 is one billion.



```rust
fn main() {
    let x = 1_000_000_000;
    println!("x: {}", x);
}
```

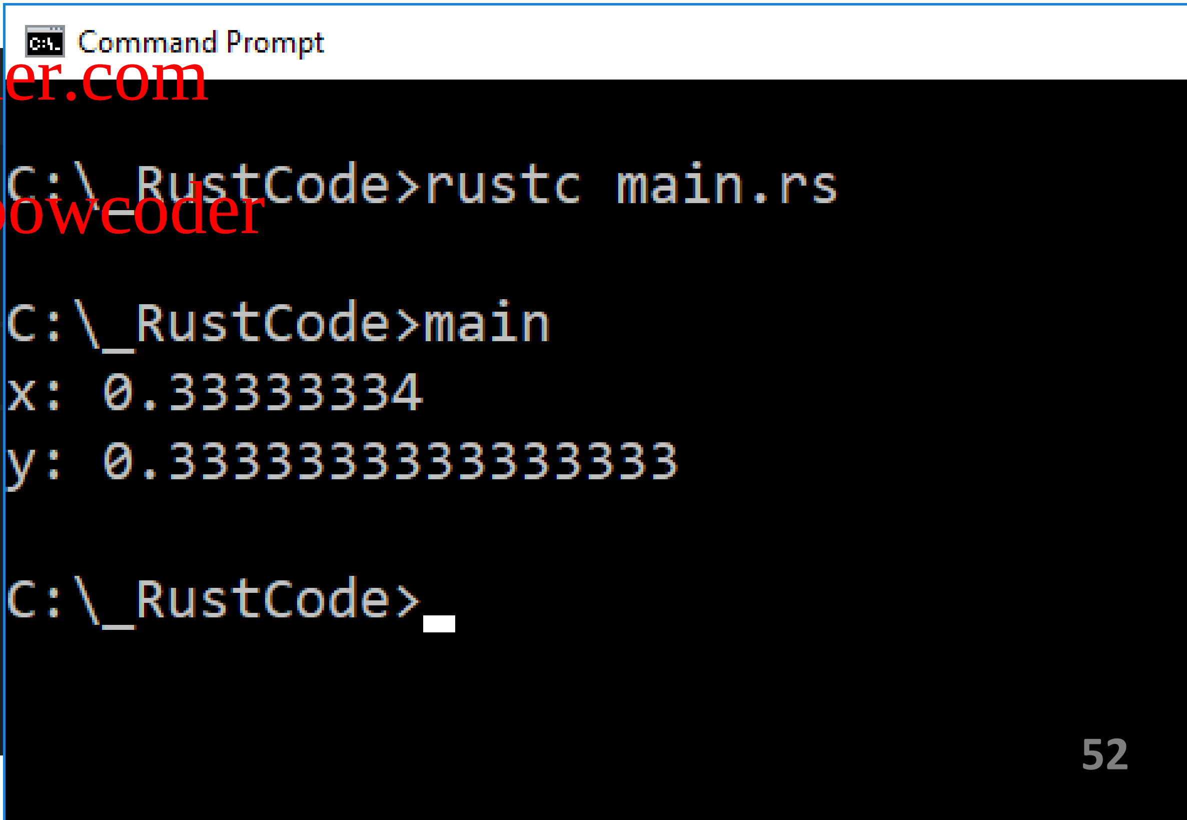
Scalar Types: Floating Point

- Two kinds – 32 and 64 bit (float and double, single and double precision)
- Represented using standard IEEE-754



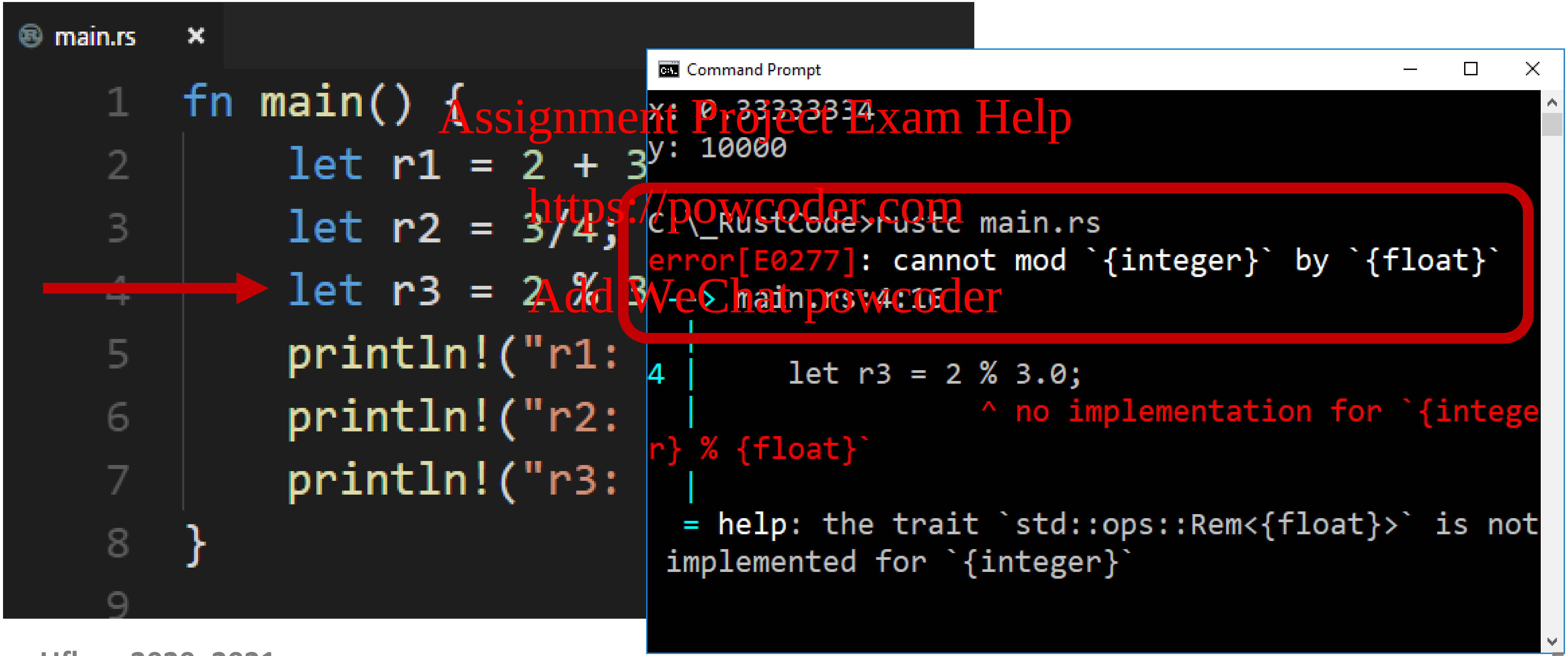
```rust
fn main() {
    let x: f32 = 1.0/3.0;
    let y: f64 = 1.0/3.0;
    println!("x: {}", x);
    println!("y: {}", y);
}
```

Default



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
x: 0.33333334
y: 0.3333333333333333

C:\_RustCode>_
```

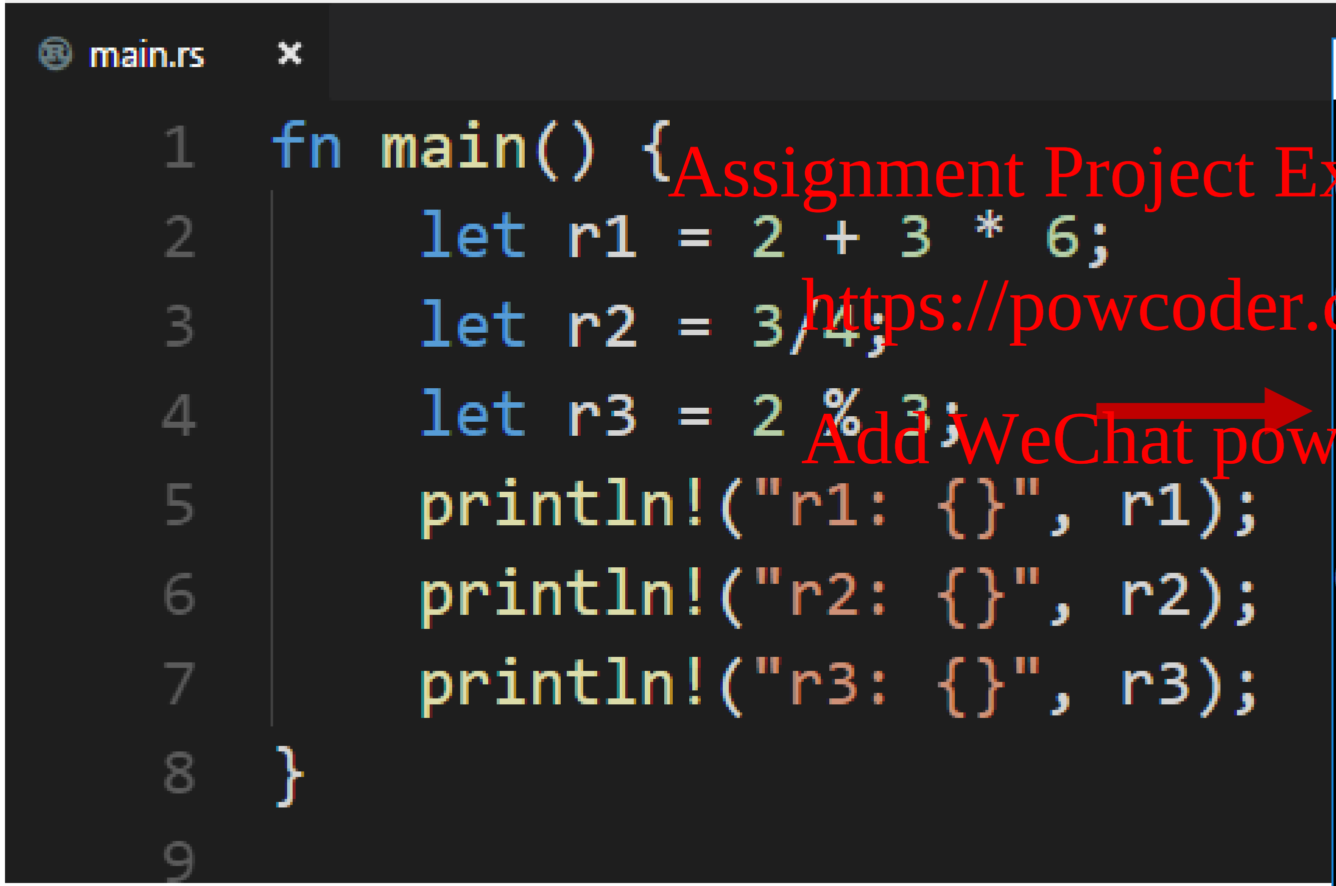
Numeric Operations



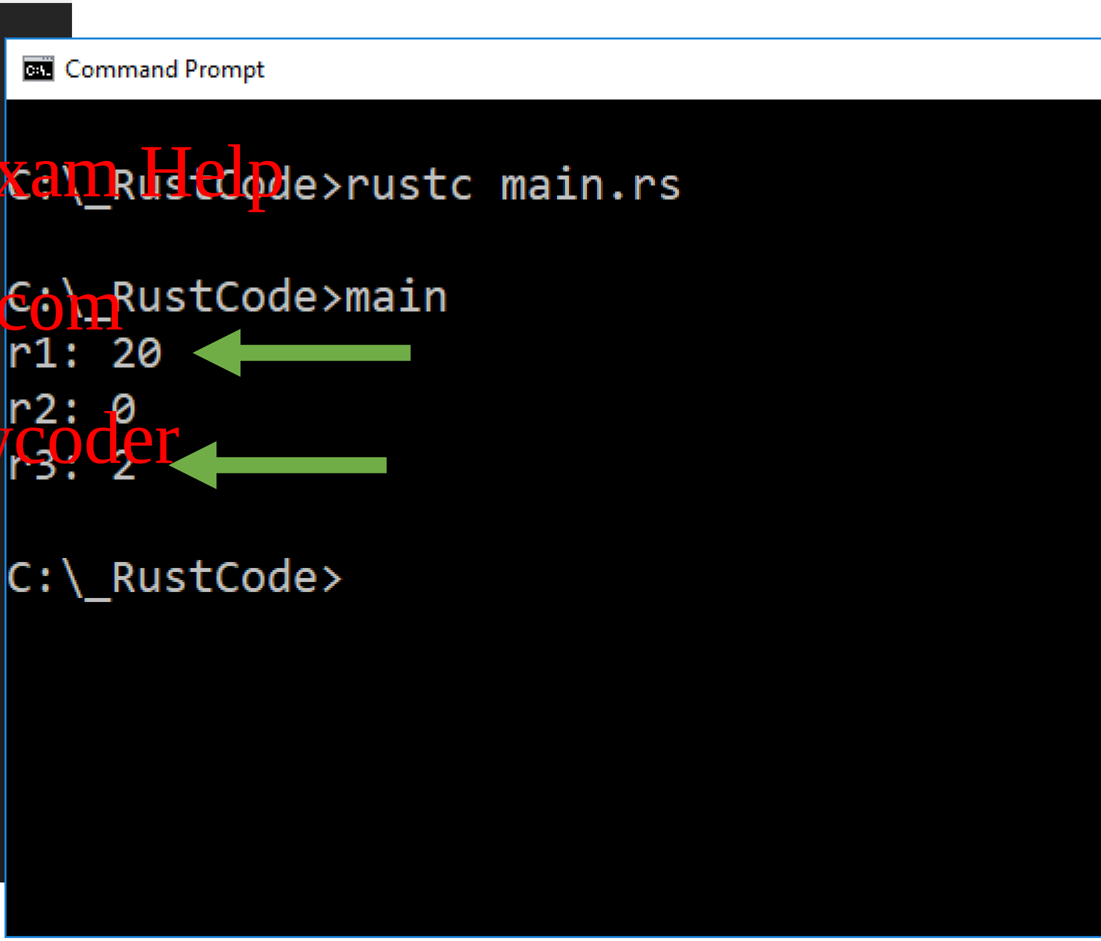
```
fn main() {
    let r1 = 2 + 3
    let r2 = 3/4;
    let r3 = 2 % 3
    println!("r1:
    println!("r2:
    println!("r3:
}
```

```
x: 0.33333334
y: 10000

C:\_RustCode>rustc main.rs
error[E0277]: cannot mod `{integer}` by `{float}`
 --> main.rs:4:15
  |
4 |     let r3 = 2 % 3.0;
  |                ^ no implementation for `{intege
r} % {float}`
  |
  = help: the trait `std::ops::Rem<{float}>` is not
 implemented for `{integer}`
```

Numeric Operations

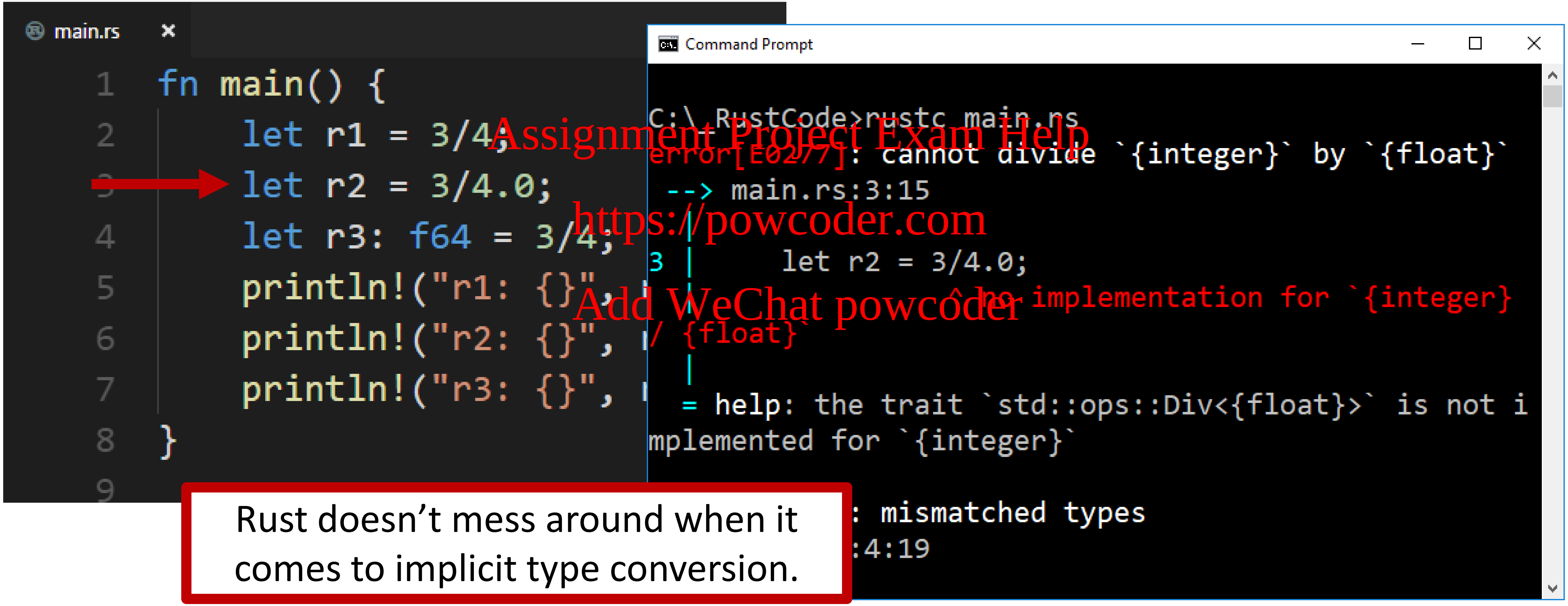


```rust
fn main() {
    let r1 = 2 + 3 * 6;
    let r2 = 3/4;
    let r3 = 2 % 3;
    println!("r1: {}", r1);
    println!("r2: {}", r2);
    println!("r3: {}", r3);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
r1: 20
r2: 0
r3: 2

C:\_RustCode>
```

Mixed Expressions?

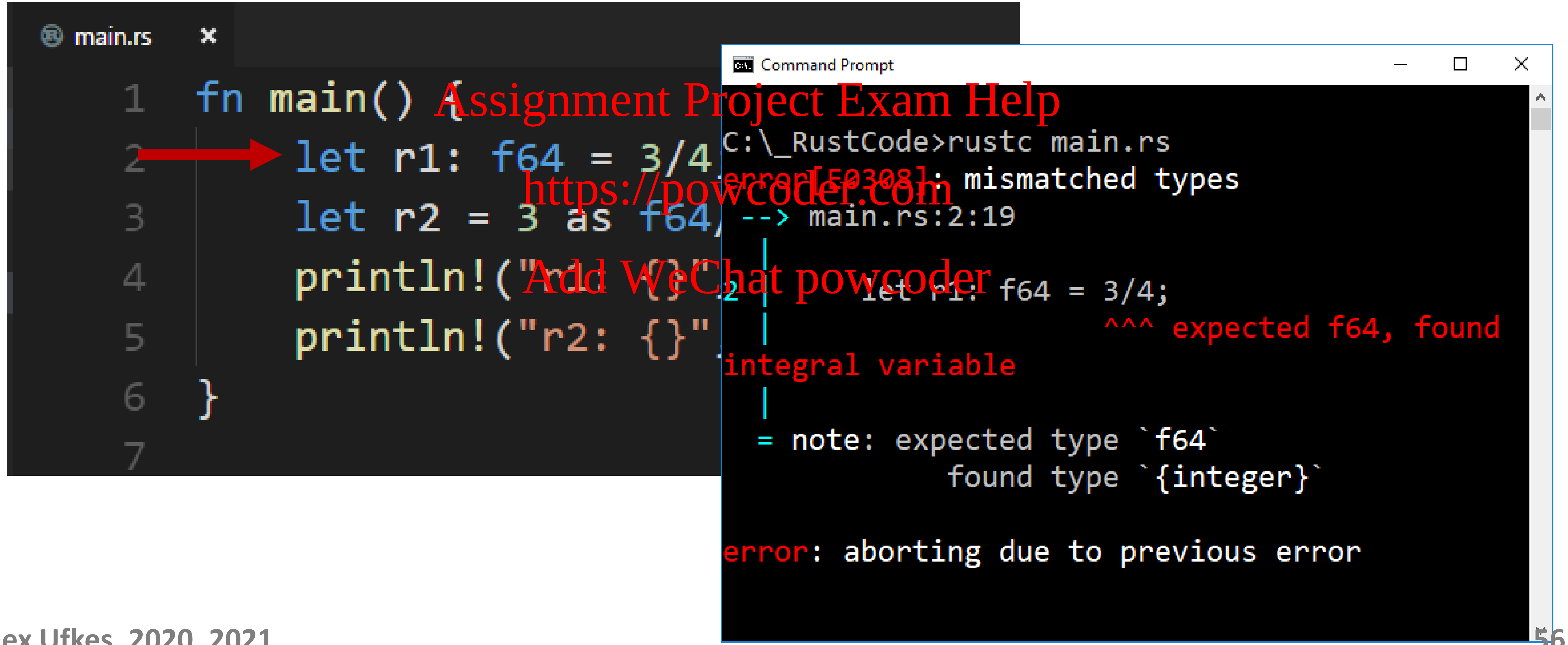


```rust
fn main() {
    let r1 = 3/4;
    let r2 = 3/4.0;
    let r3: f64 = 3/4;
    println!("r1: {}",
    println!("r2: {}",
    println!("r3: {}",
}
```

```
C:\RustCode>rustc main.rs
error[E0277]: cannot divide `{integer}` by `{float}`
 --> main.rs:3:15
  |
3 |     let r2 = 3/4.0;
  |               ^ no implementation for `{integer}
 / {float}`
  |
  = help: the trait `std::ops::Div<{float}>` is not i
mplemented for `{integer}`

                 : mismatched types
                 :4:19
```

Rust doesn't mess around when it comes to implicit type conversion.

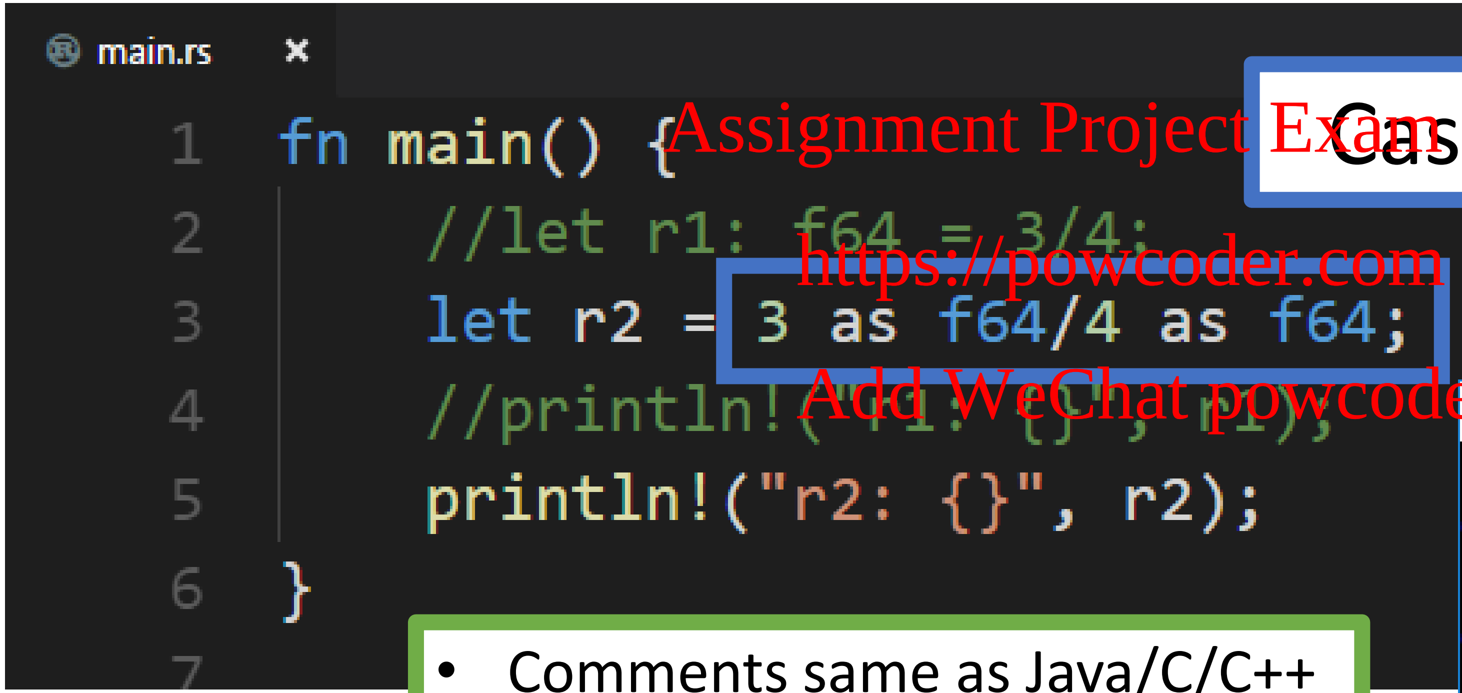
Mixed Expressions?



```
fn main() {
    let r1: f64 = 3/4
    let r2 = 3 as f64
    println!("r1: {}"
    println!("r2: {}"
}
```

```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:2:19
  |
2 |     let r1: f64 = 3/4;
  |                   ^^^ expected f64, found
integral variable
  |
  = note: expected type `f64`
             found type `{integer}`

error: aborting due to previous error
```

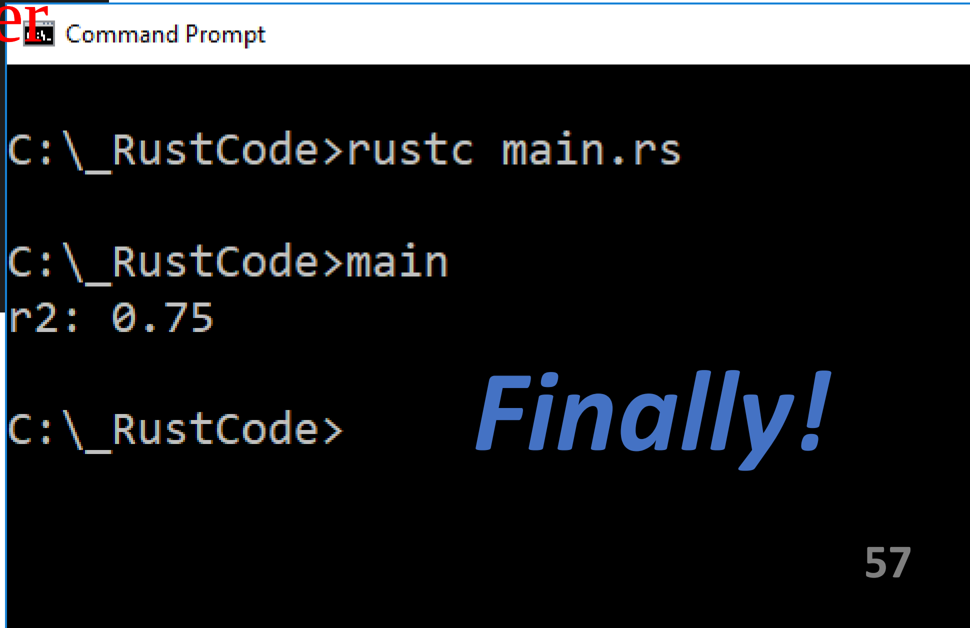
Mixed Expressions?



```
fn main() {
    //let r1: f64 = 3/4;
    let r2 = 3 as f64/4 as f64;
    //println!("r1: {}", r1);
    println!("r2: {}", r2);
}
```

Cast using: as *type*

- Comments same as Java/C/C++
- Both block and single-line

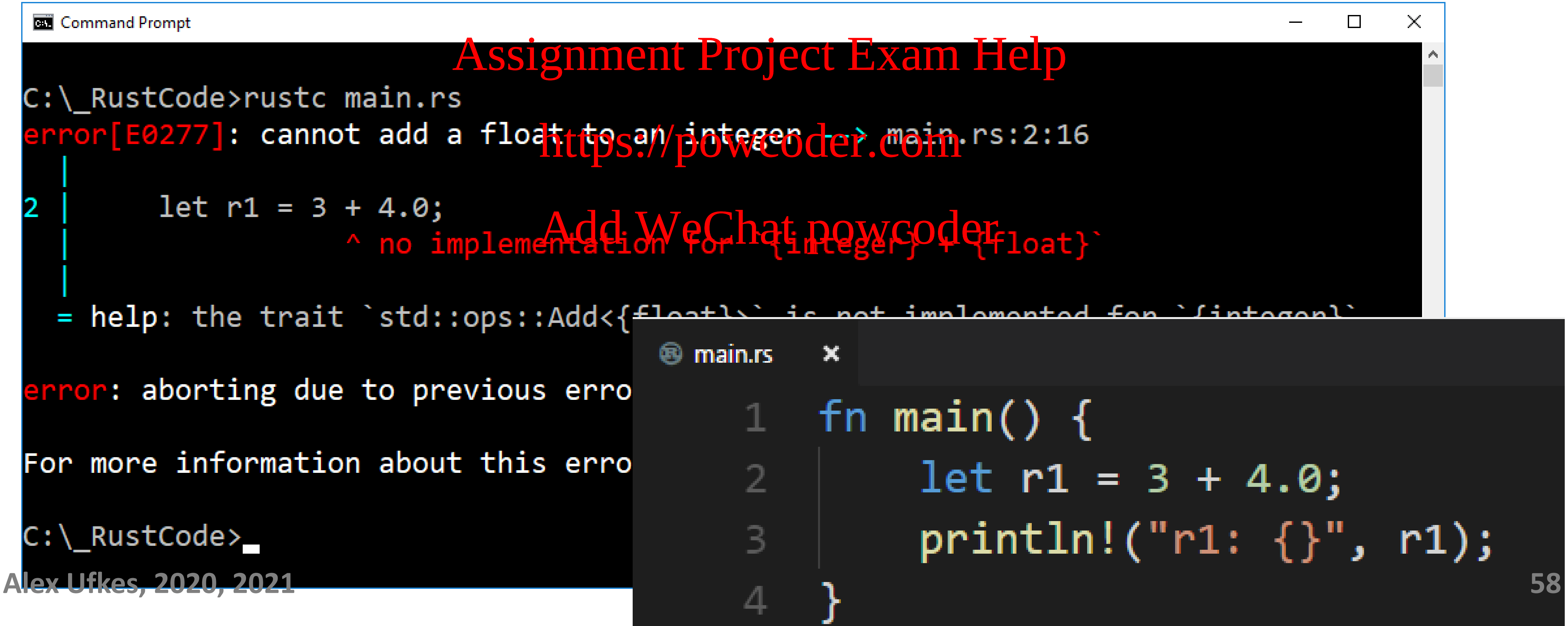


```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
r2: 0.75

C:\_RustCode>
```

Finally!

Mixed Expressions?

Division may truncate, good reason to avoid implicit conversion…



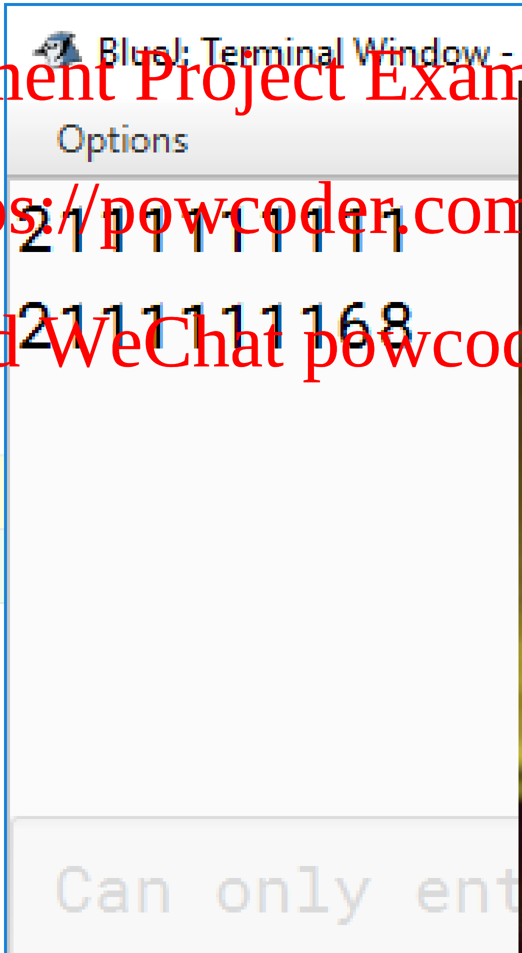
```
C:\_RustCode>rustc main.rs
error[E0277]: cannot add a float to an integer --> main.rs:2:16
  |
2 |     let r1 = 3 + 4.0;
  |                ^ no implementation for `{integer} + {float}`
  |
  = help: the trait `std::ops::Add<{float}>` is not implemented for `{integer}`

error: aborting due to previous erro

For more information about this erro

C:\_RustCode>_
```

```rust
fn main() {
    let r1 = 3 + 4.0;
    println!("r1: {}", r1);
}
```

Why?!

- Adding `float` to `int` means converting the integer to a floating-point type, then adding.
- CPU doesn't add different types.
- Float and int arithmetic is done using different instructions, in different locations on CPU.
- It's possible to introduce errors in precision!
- An integer in binary is *exactly precise*.
- The same value represented as a floating point may lose significant digits.
- Most languages don't even warn about this – Rust doesn't allow it at all.

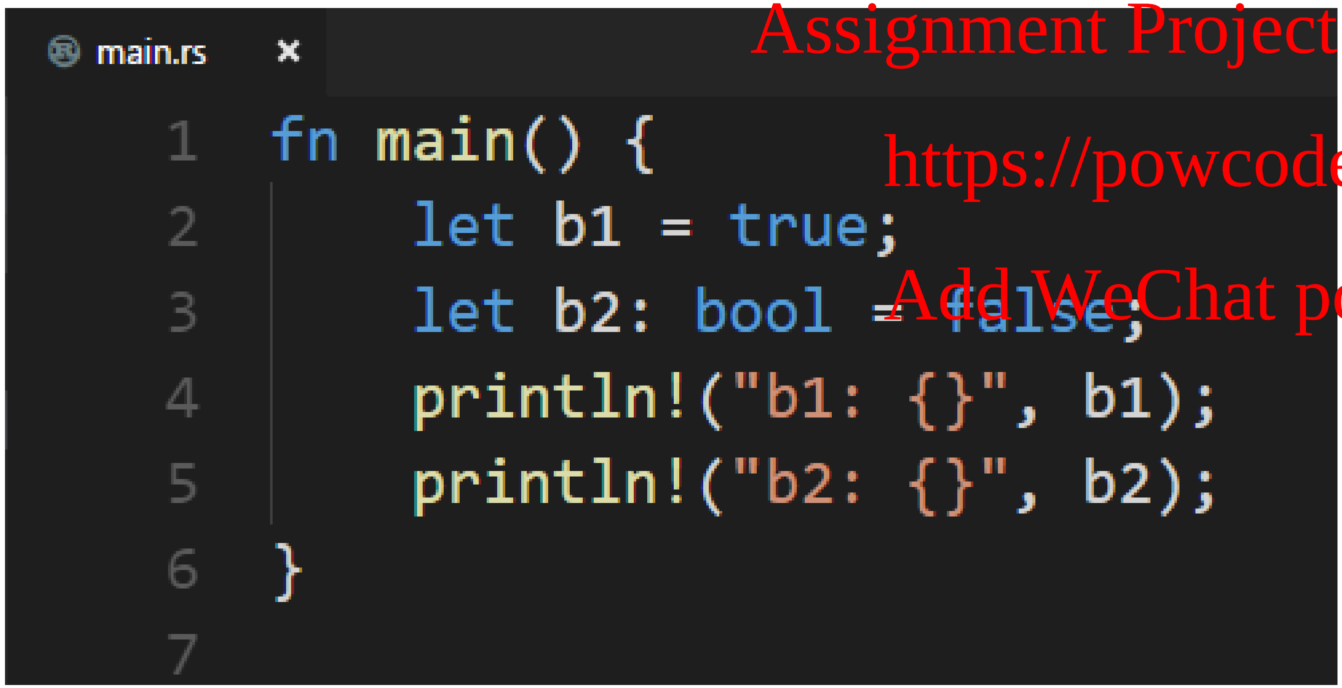
```java
public class MethodTester
{
    public static void main(String[] args)
    {
        int a = 2111111111;
        System.out.println(a);

        float b = a;
        a = (int) b;

        System.out.println(a);
    }
}
```



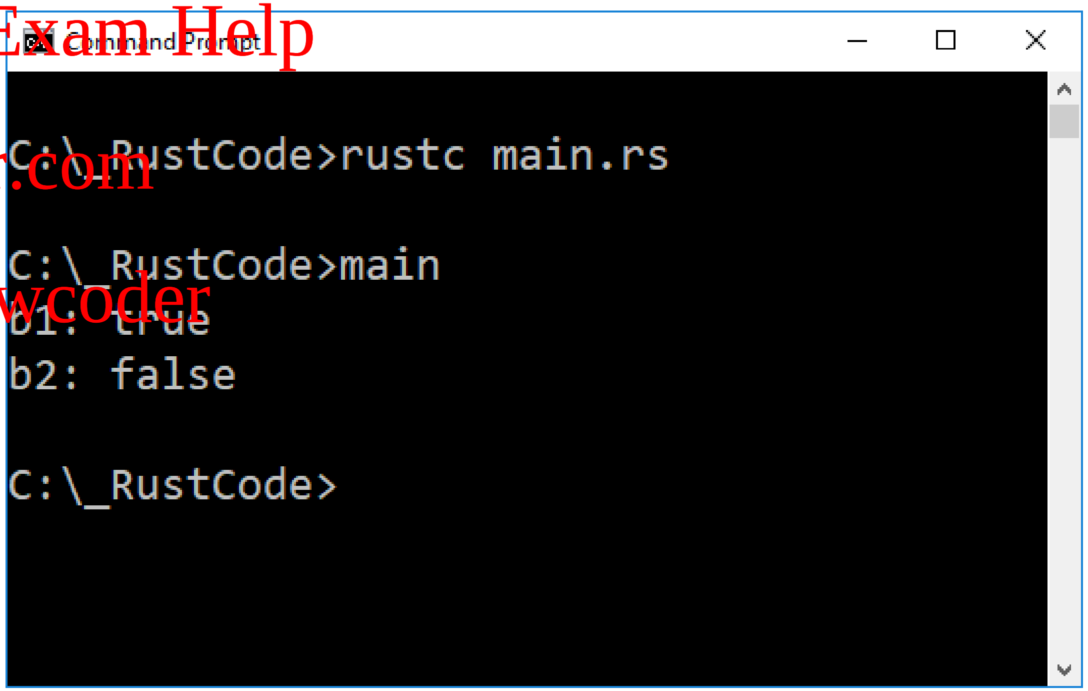
```
2111111111
2111111168
```

Scalar Types: Boolean

`true`, `false`. Easy:



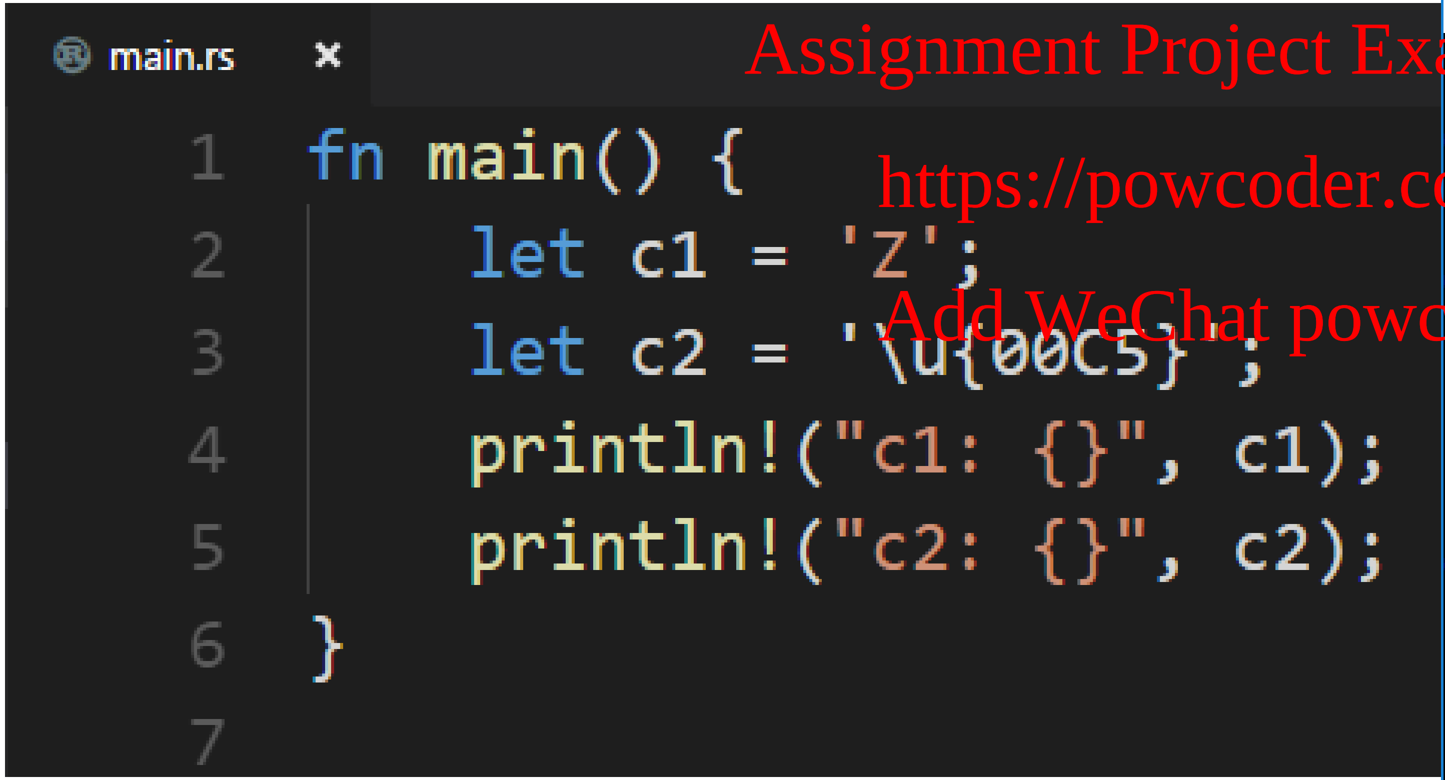
```rust
fn main() {
    let b1 = true;
    let b2: bool = false;
    println!("b1: {}", b1);
    println!("b2: {}", b2);
}
```



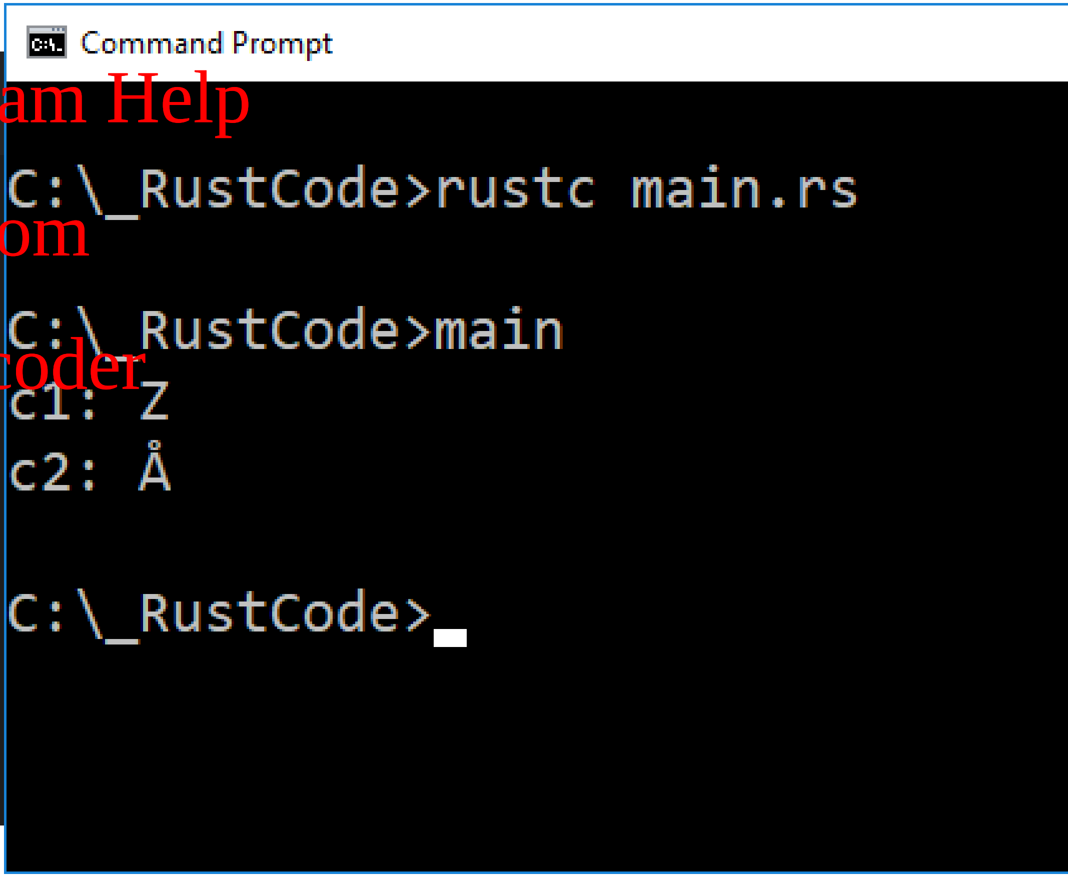
```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
b1: true
b2: false

C:\_RustCode>
```

Scalar Types: Characters

Rust supports Unicode:

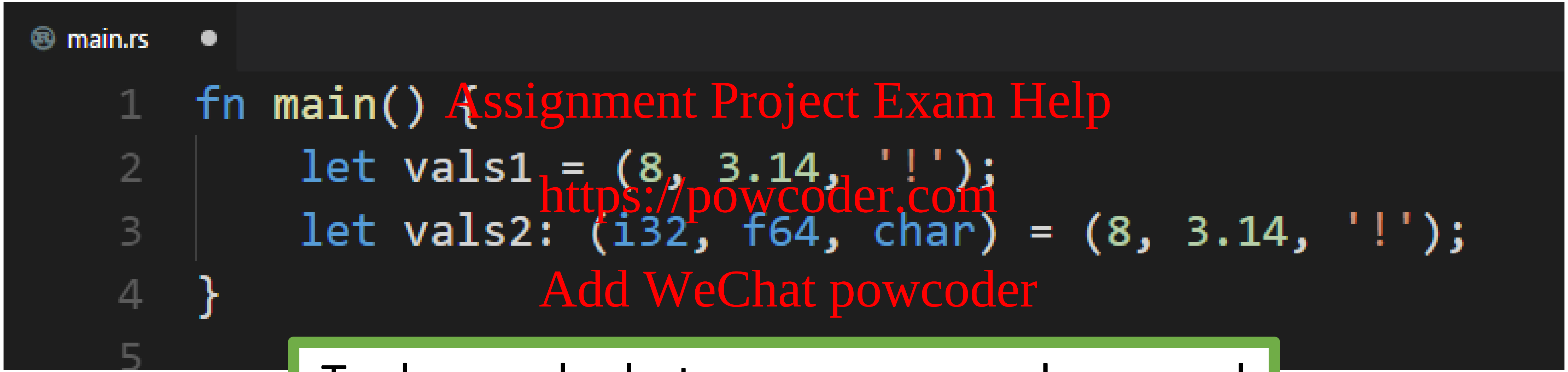


```rust
fn main() {
    let c1 = 'Z';
    let c2 = '\u{00C5}';
    println!("c1: {}", c1);
    println!("c2: {}", c2);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
c1: Z
c2: Å

C:\_RustCode>
```

Compound Types: Tuples

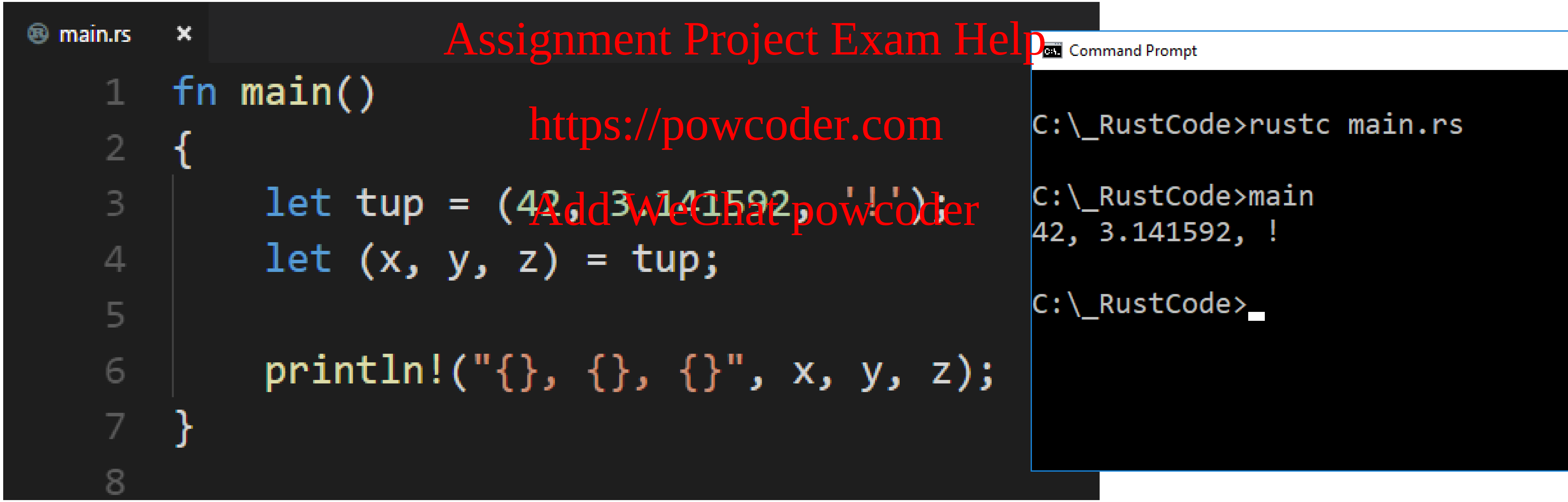


```rust
fn main() {
    let vals1 = (8, 3.14, '!');
    let vals2: (i32, f64, char) = (8, 3.14, '!');
}
```

Tuples can be heterogeneous, and we need not specify type. Rust can infer it.

Accessing Elements

De-structuring!

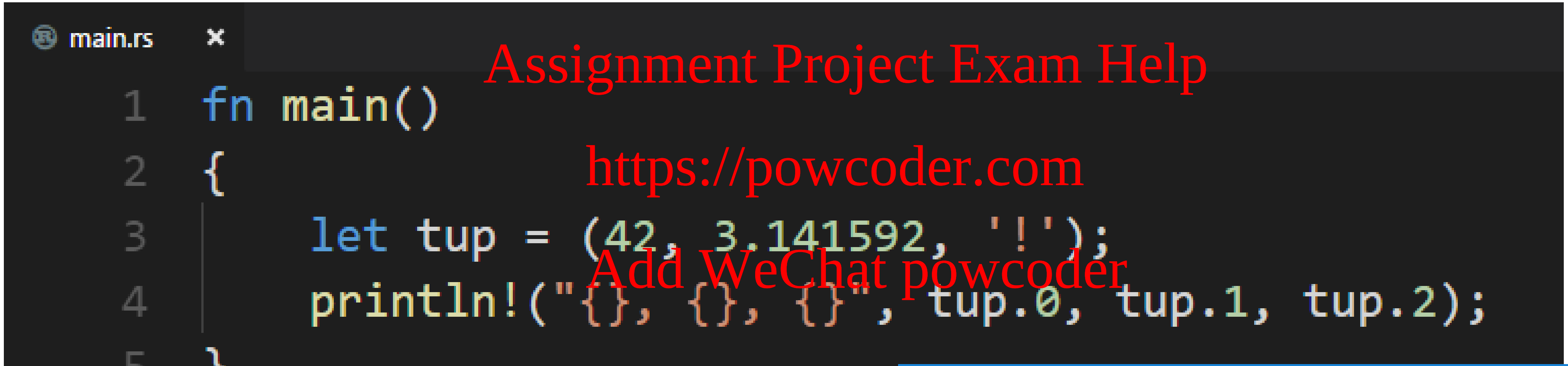


```rust
fn main()
{
    let tup = (42, 3.141592, '!');
    let (x, y, z) = tup;

    println!("{}, {}, {}", x, y, z);
}
```

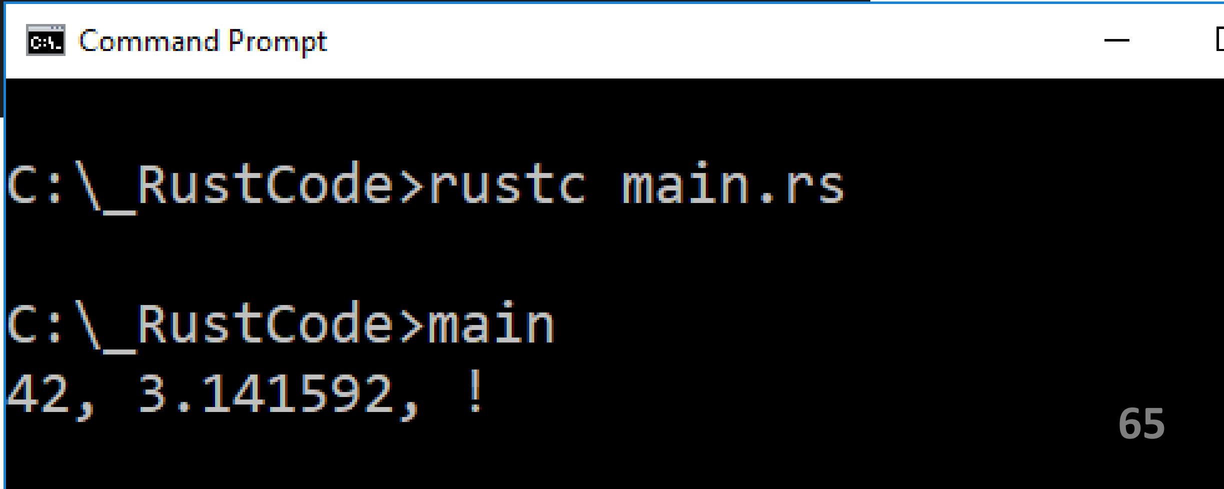
```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
42, 3.141592, !

C:\_RustCode>_
```

Accessing Elements

Can also access directly:



```
fn main()
{
    let tup = (42, 3.141592, '!');
    println!("{}, {}, {}", tup.0, tup.1, tup.2);
}
```

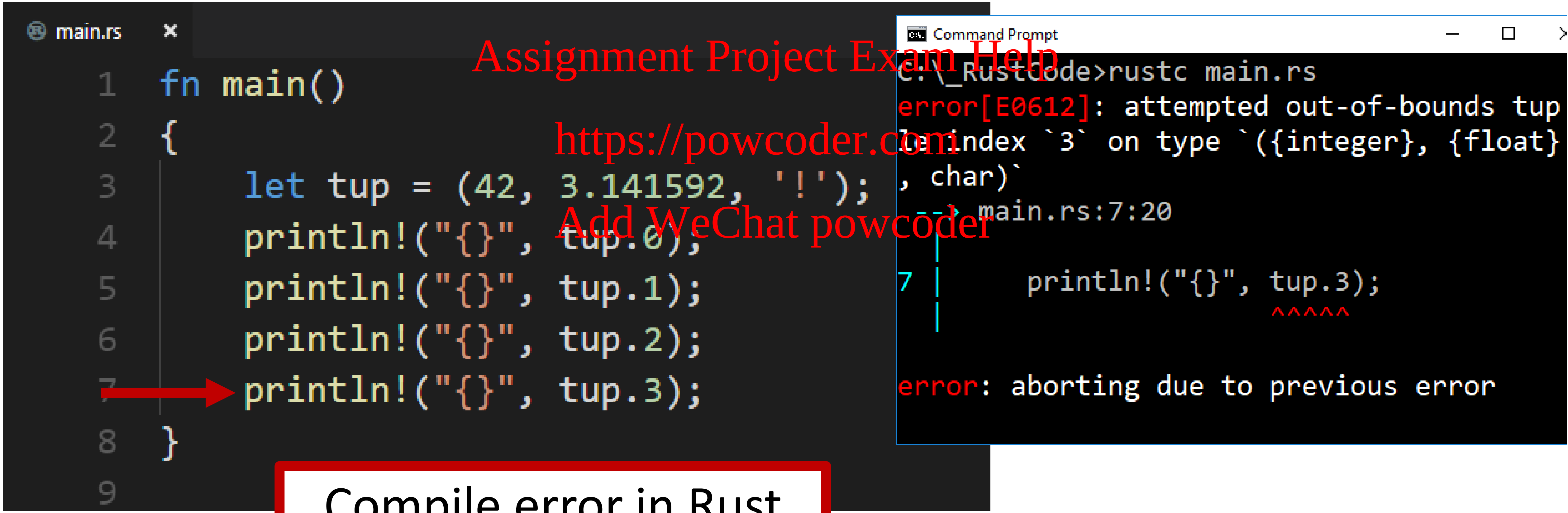


```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
42, 3.141592, !
```

Can we go out of bounds?

Accessing Elements

Out of bounds:



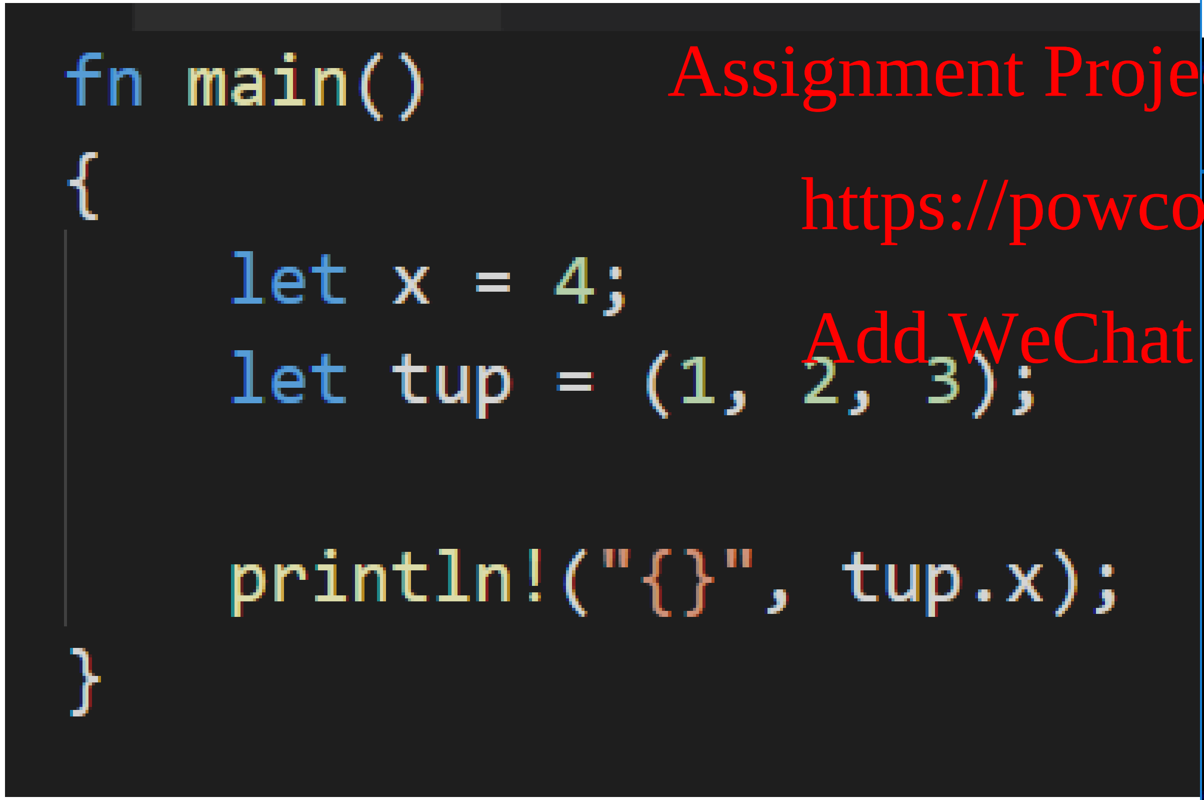
```
fn main()
{
    let tup = (42, 3.141592, '!');
    println!("{}", tup.0);
    println!("{}", tup.1);
    println!("{}", tup.2);
    println!("{}", tup.3);
}
```

```
C:\_RustCode>rustc main.rs
error[E0612]: attempted out-of-bounds tup
le index `3` on type `({integer}, {float}
, char)`
 --> main.rs:7:20
  |
7 |     println!("{}", tup.3);
  |                    ^^^^^

error: aborting due to previous error
```

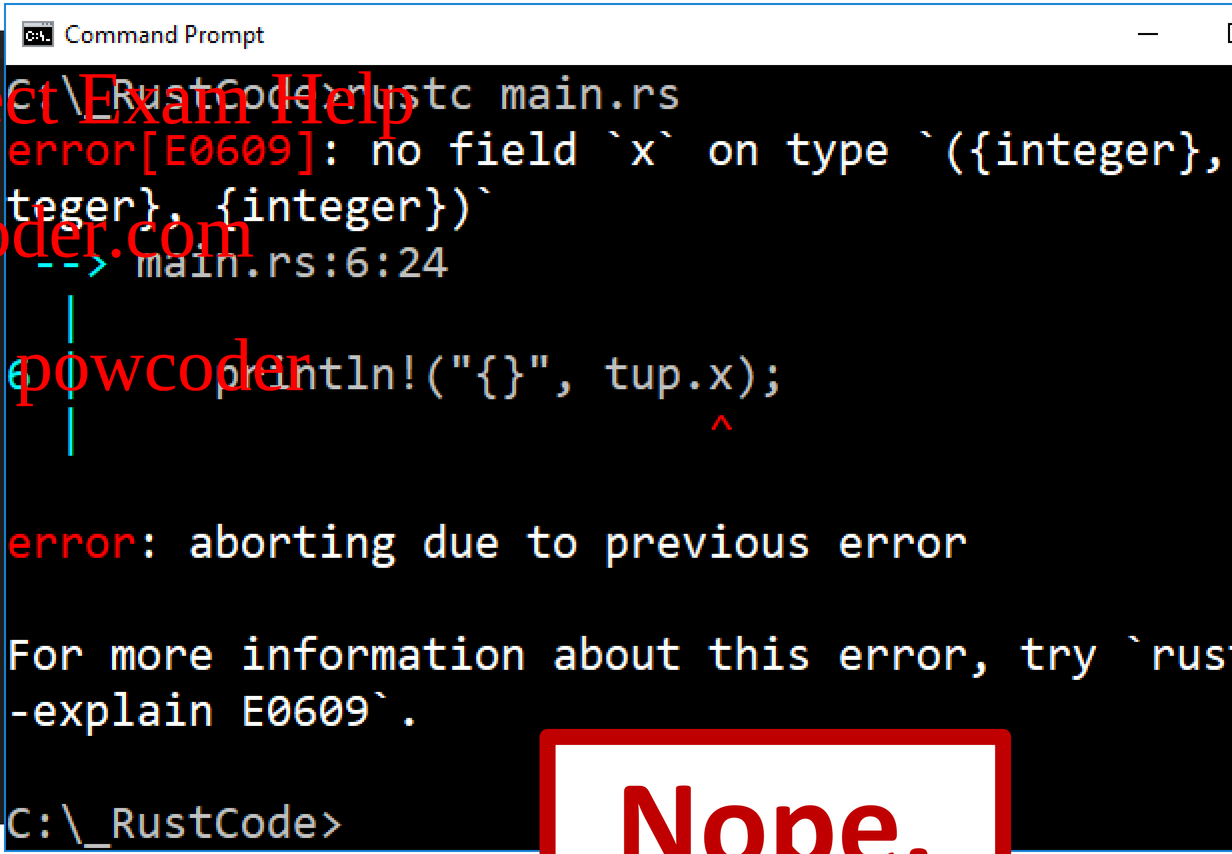
Compile error in Rust

Accessing Elements

Can we fool it?



```
fn main()
{
    let x = 4;
    let tup = (1, 2, 3);

    println!("{}", tup.x);
}
```

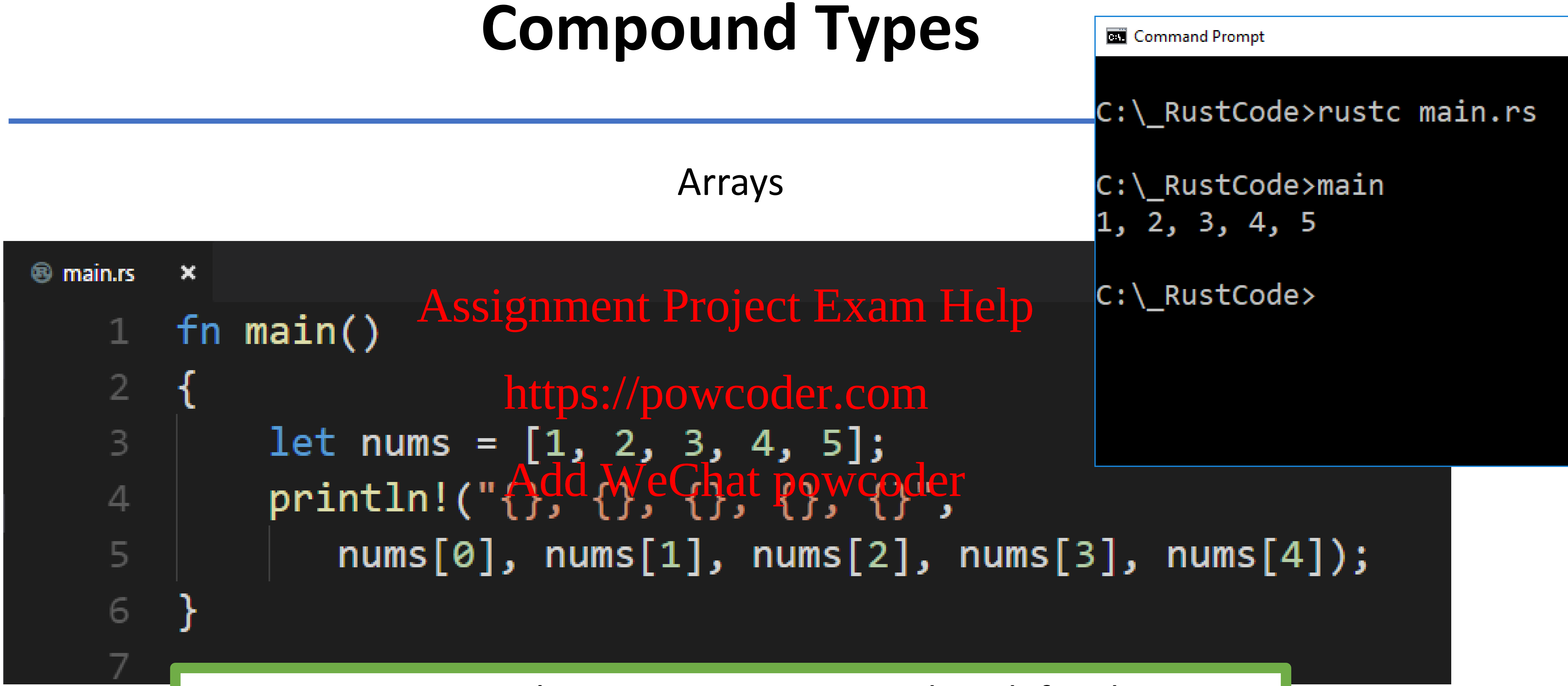


```
C:\_RustCode>rustc main.rs
error[E0609]: no field `x` on type `({integer},
teger}, {integer})`
 --> main.rs:6:24
  |
6 |     println!("{}", tup.x);
  |                        ^

error: aborting due to previous error

For more information about this error, try `rus
-explain E0609`.

C:\_RustCode>
```

Nope.

Compound Types

Arrays



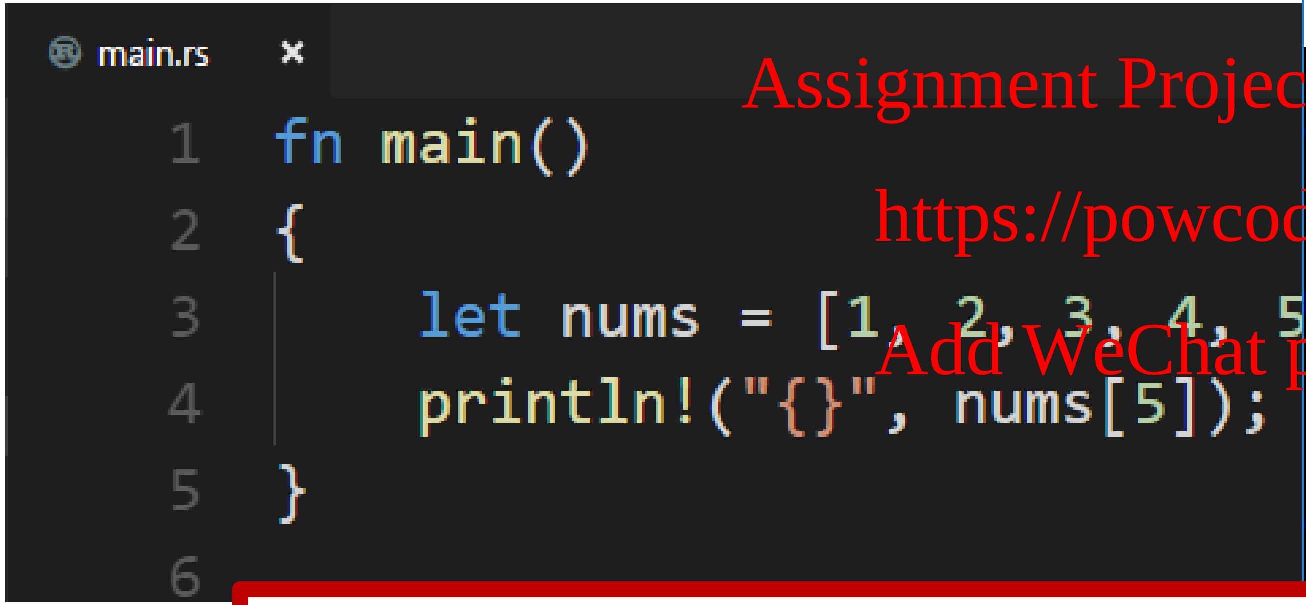
```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5];
    println!("{}, {}, {}, {}, {}",
        nums[0], nums[1], nums[2], nums[3], nums[4]);
}
```

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
1, 2, 3, 4, 5

C:\_RustCode>
```

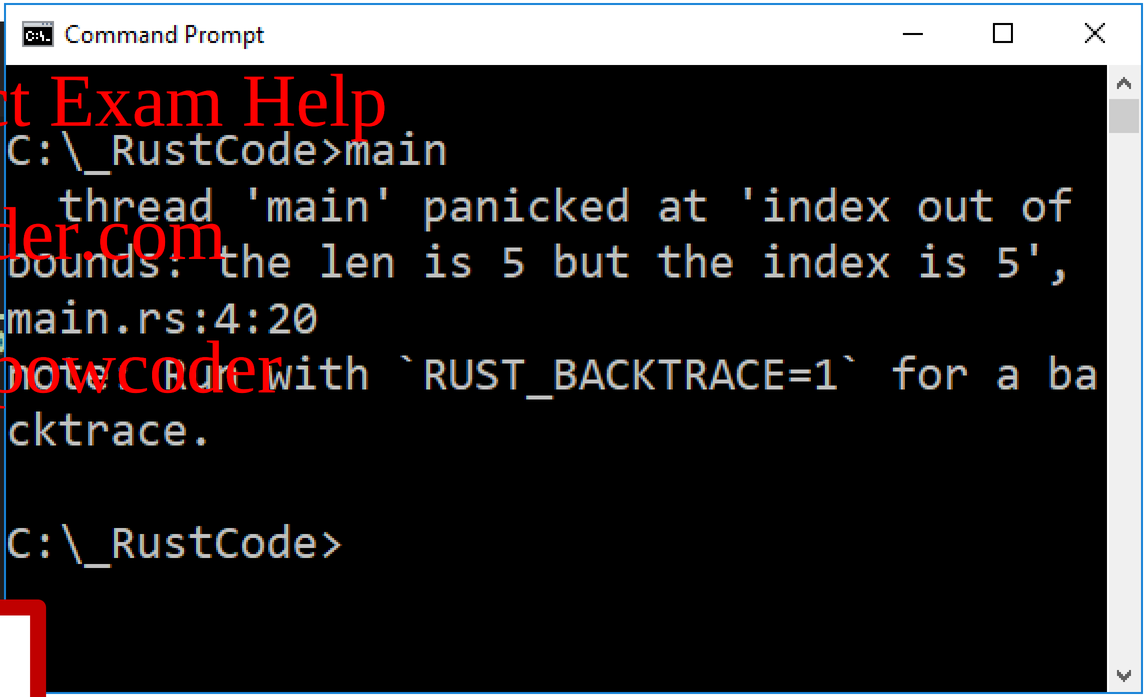
Arrays in Rust are: homogeneous, zero-indexed, fixed in size.

Accessing Elements

Out of bounds:



```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5
    println!("{}", nums[5]);
}
```

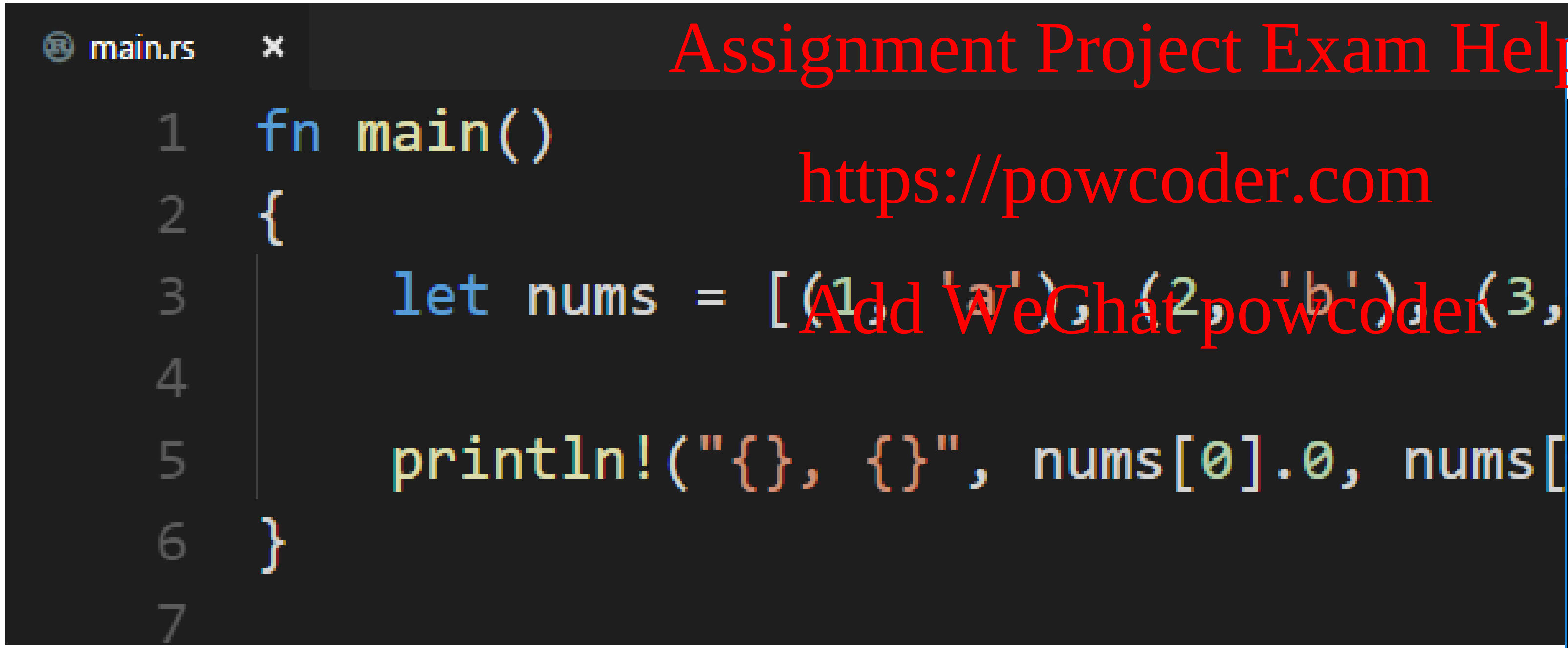


```
C:\_RustCode>main
  thread 'main' panicked at 'index out of
bounds: the len is 5 but the index is 5',
main.rs:4:20
note: Run with `RUST_BACKTRACE=1` for a ba
cktrace.

C:\_RustCode>
```

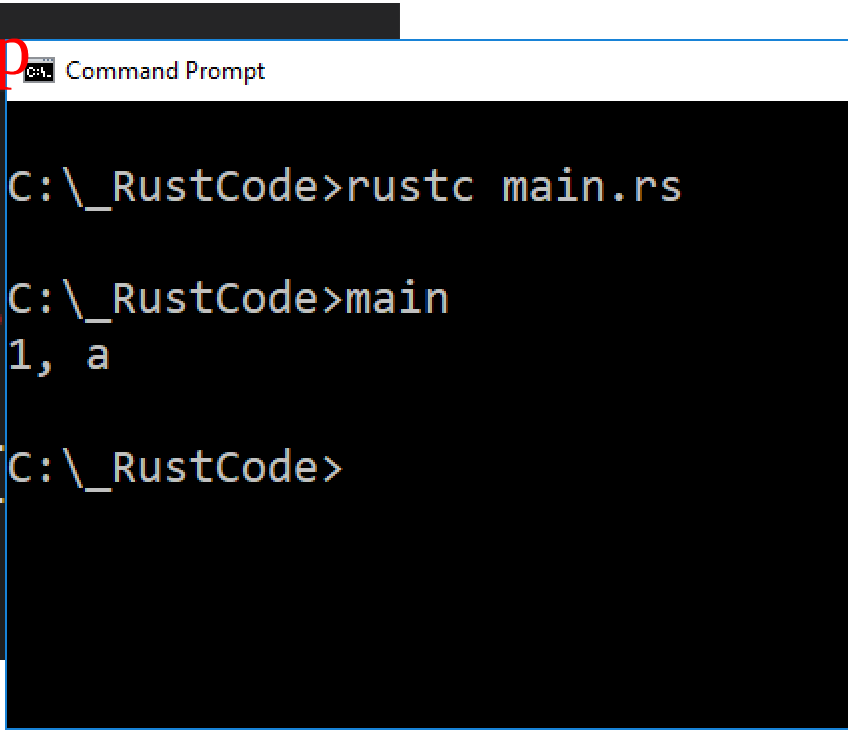
Runtime error, much like Java.
Prevents out of bounds array accesses.

Array of Tuples

Same rules as Haskell:



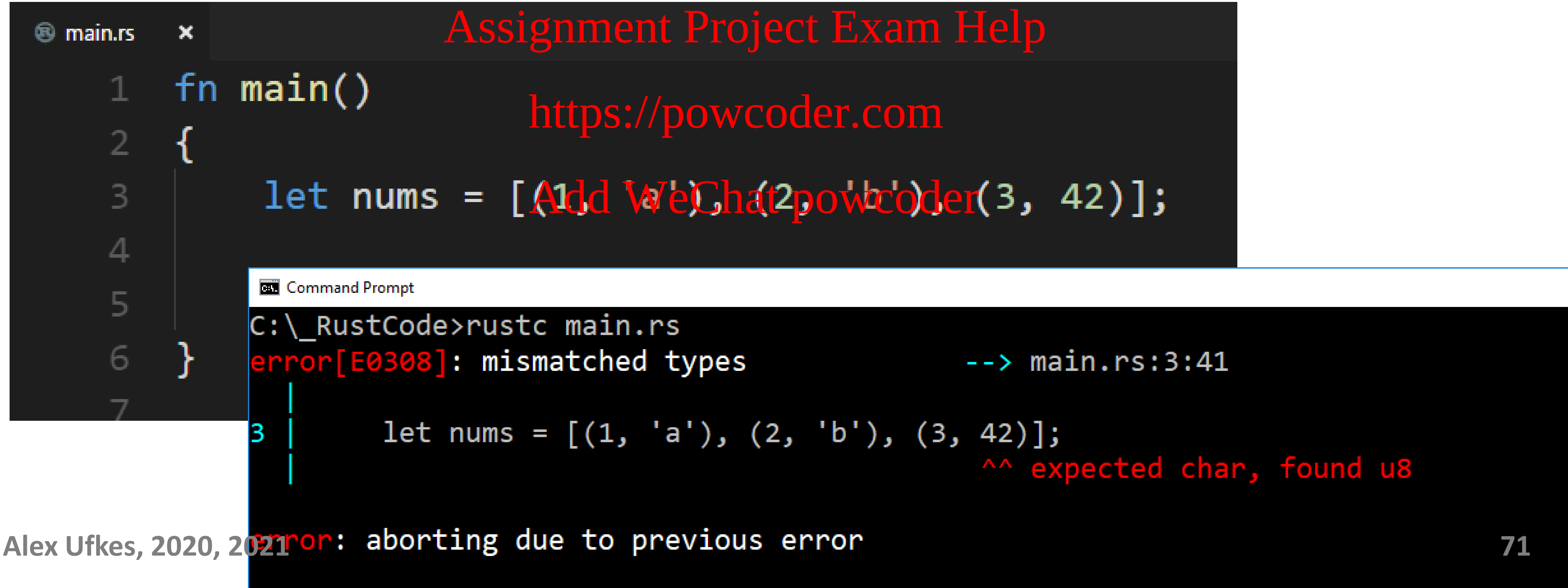
```rust
fn main()
{
    let nums = [(1, 'a'), (2, 'b'), (3,

    println!("{}, {}", nums[0].0, nums[
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
1, a

C:\_RustCode>
```

Array of Tuples

Same rules as Haskell: Tuple types must be the same



```rust
fn main()
{
    let nums = [(1, 'a'), (2, 'b'), (3, 42)];


}
```

```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types                    --> main.rs:3:41
  |
3 |     let nums = [(1, 'a'), (2, 'b'), (3, 42)];
  |                                         ^^ expected char, found u8

error: aborting due to previous error
```

Types & Literals: Summary

4 Scalar types:

Integer – `u8, u16, u32, u64, usize, i8, i16,` **`i32`**`, i64, isize`

Floating Point – `f32,` **`f64`**

Boolean – `bool` (true, false)

Character – Unicode: `‘Z’, ‘a’, ‘&’, ‘\u{00C5}’,` etc

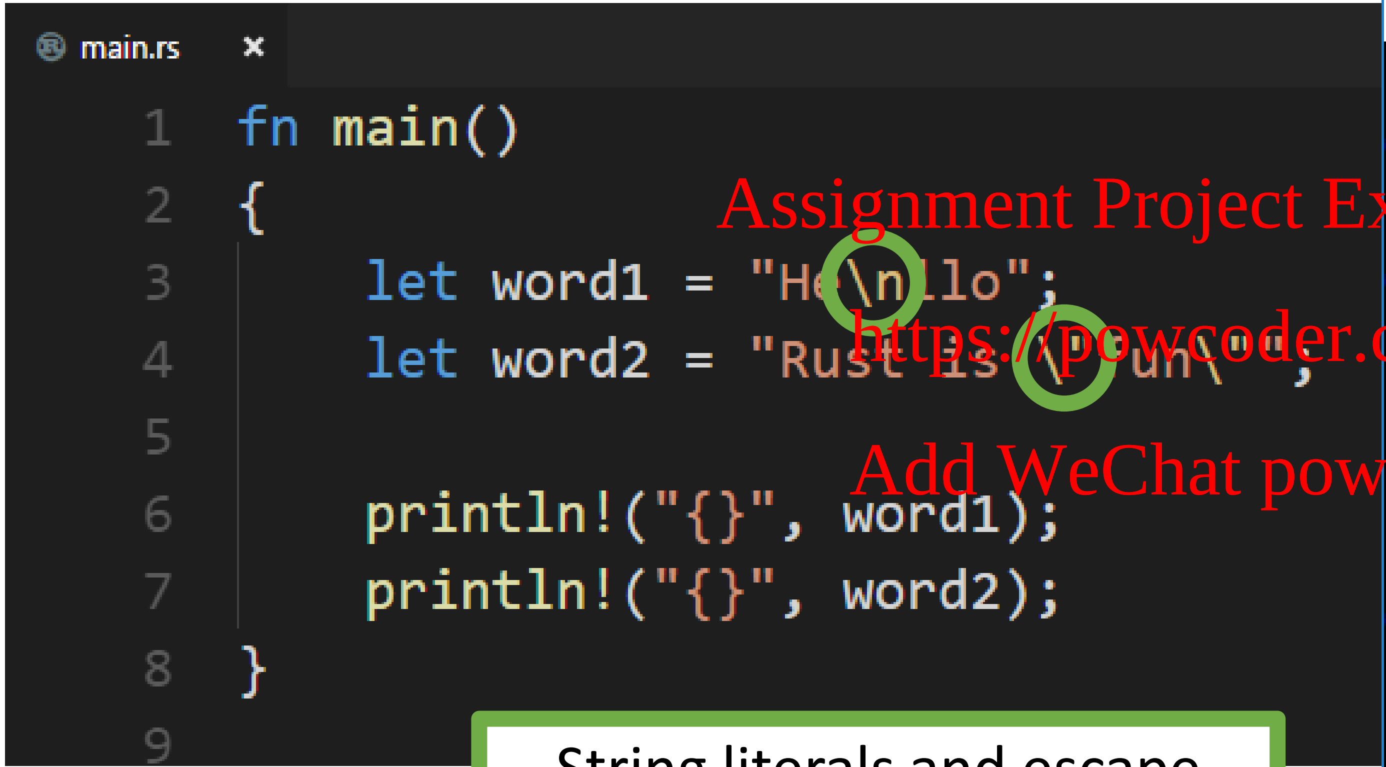
2 Compound types:

Tuple – heterogeneous

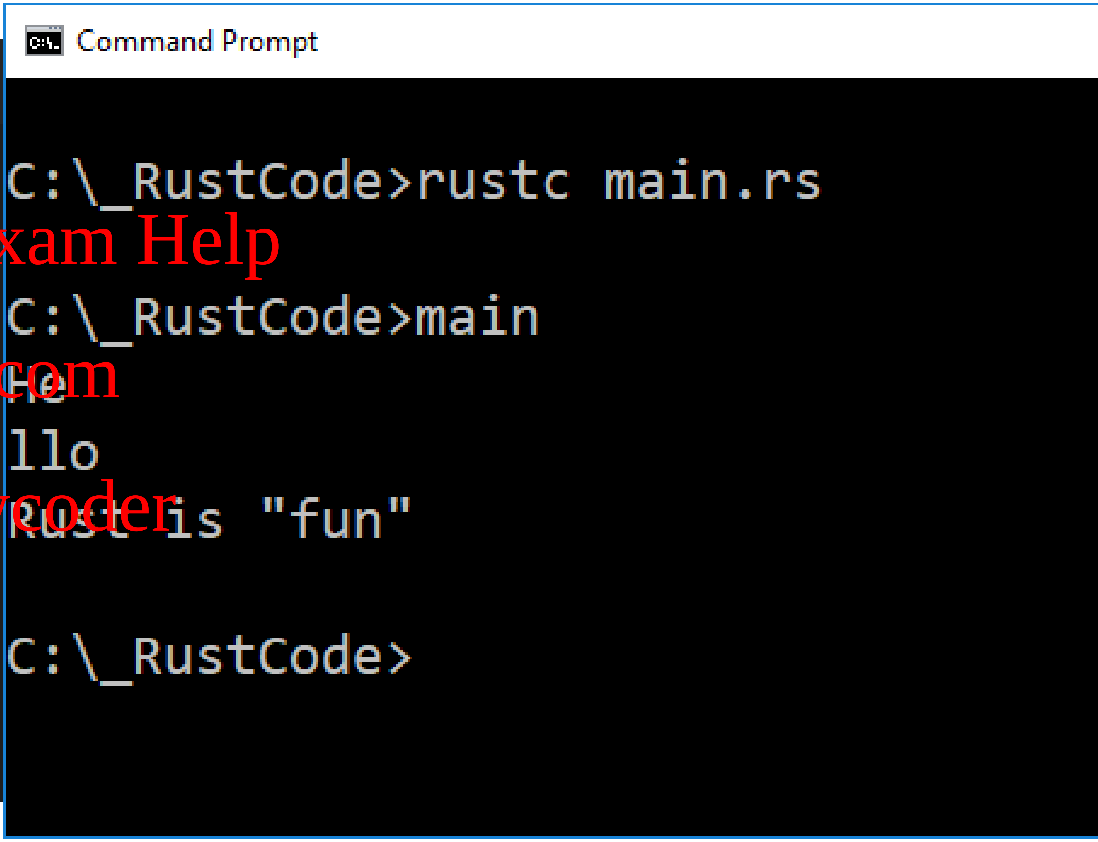
Arrays – homogeneous

Rust supports other data structures such as strings and vectors. These are not base types, but very useful.

Strings



```
fn main()
{
    let word1 = "He\nllo";
    let word2 = "Rust is \"fun\"";

    println!("{}", word1);
    println!("{}", word2);
}
```



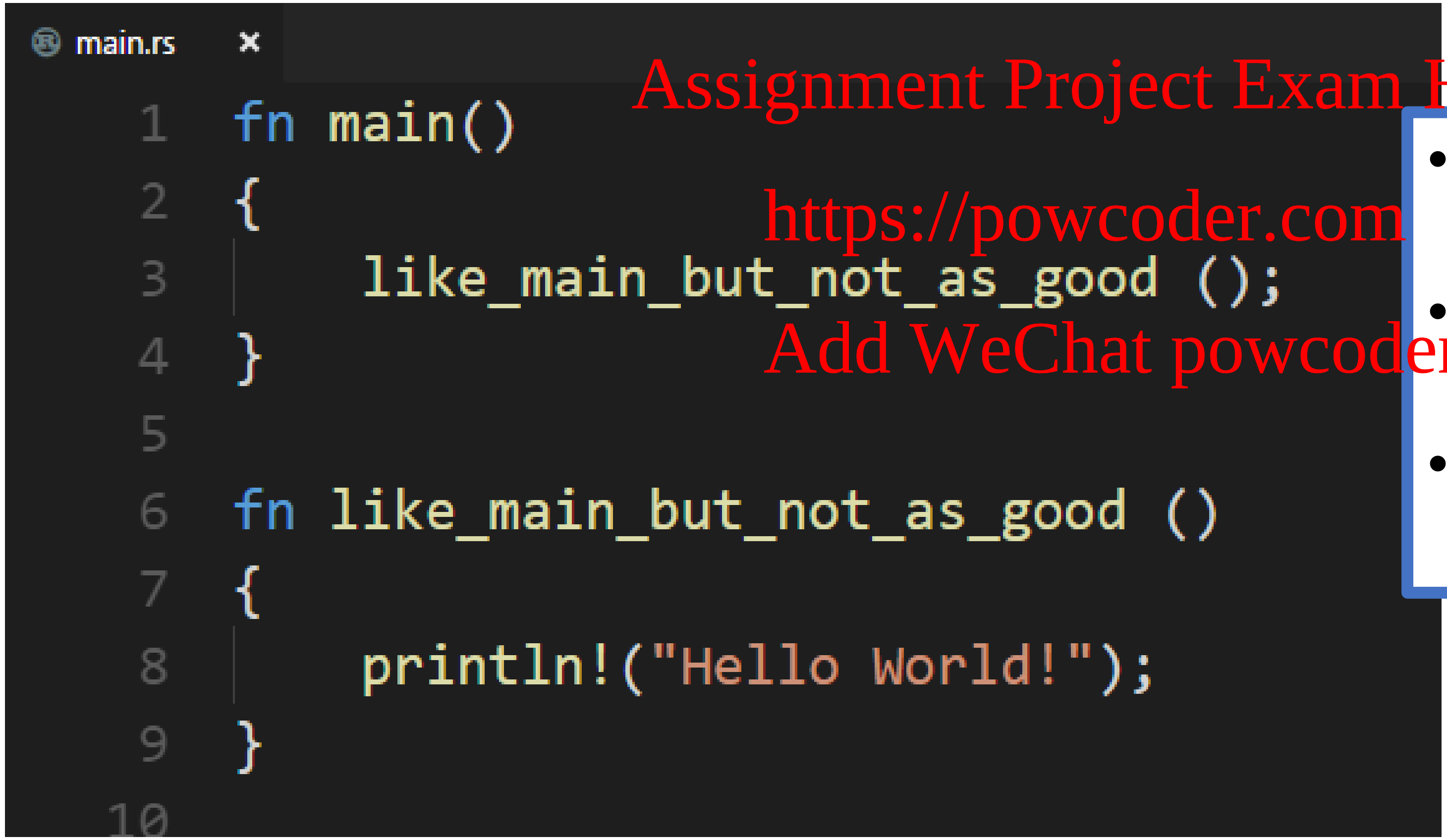
```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
He
llo
Rust is "fun"

C:\_RustCode>
```

String literals and escape characters are as expected

s

Functions

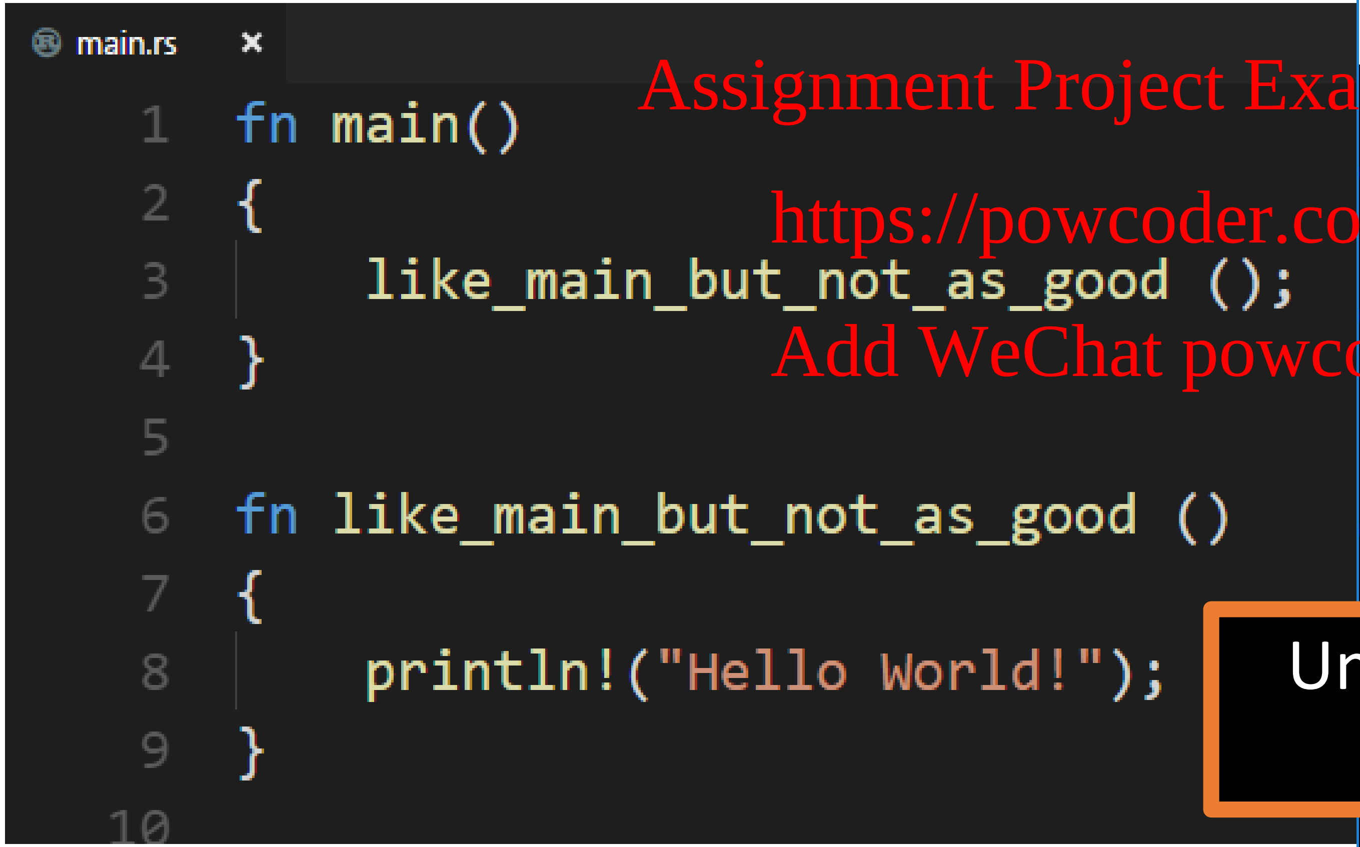
We’ve seen `main()`



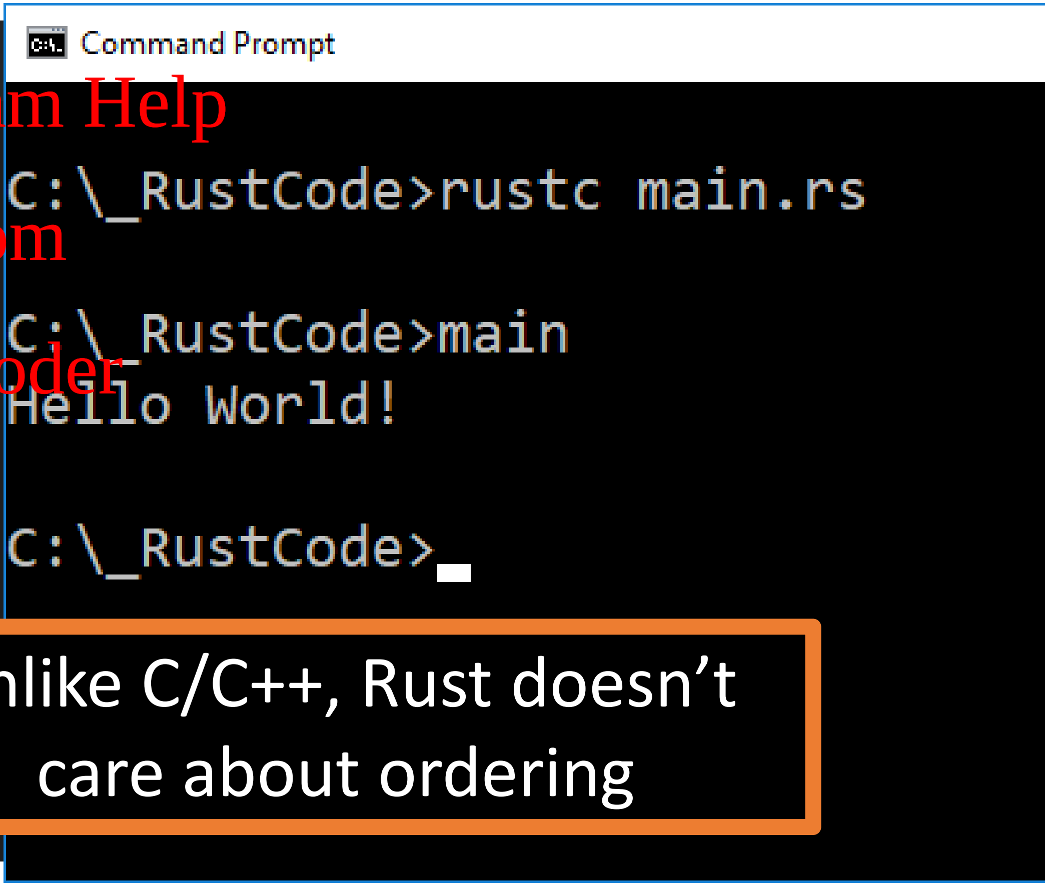
```rust
fn main()
{
    like_main_but_not_as_good ();
}

fn like_main_but_not_as_good ()
{
    println!("Hello World!");
}
```

- Returns nothing, accepts no arguments.
- Convention for naming functions is snake_case.
- Words separated by underscores.

Functions



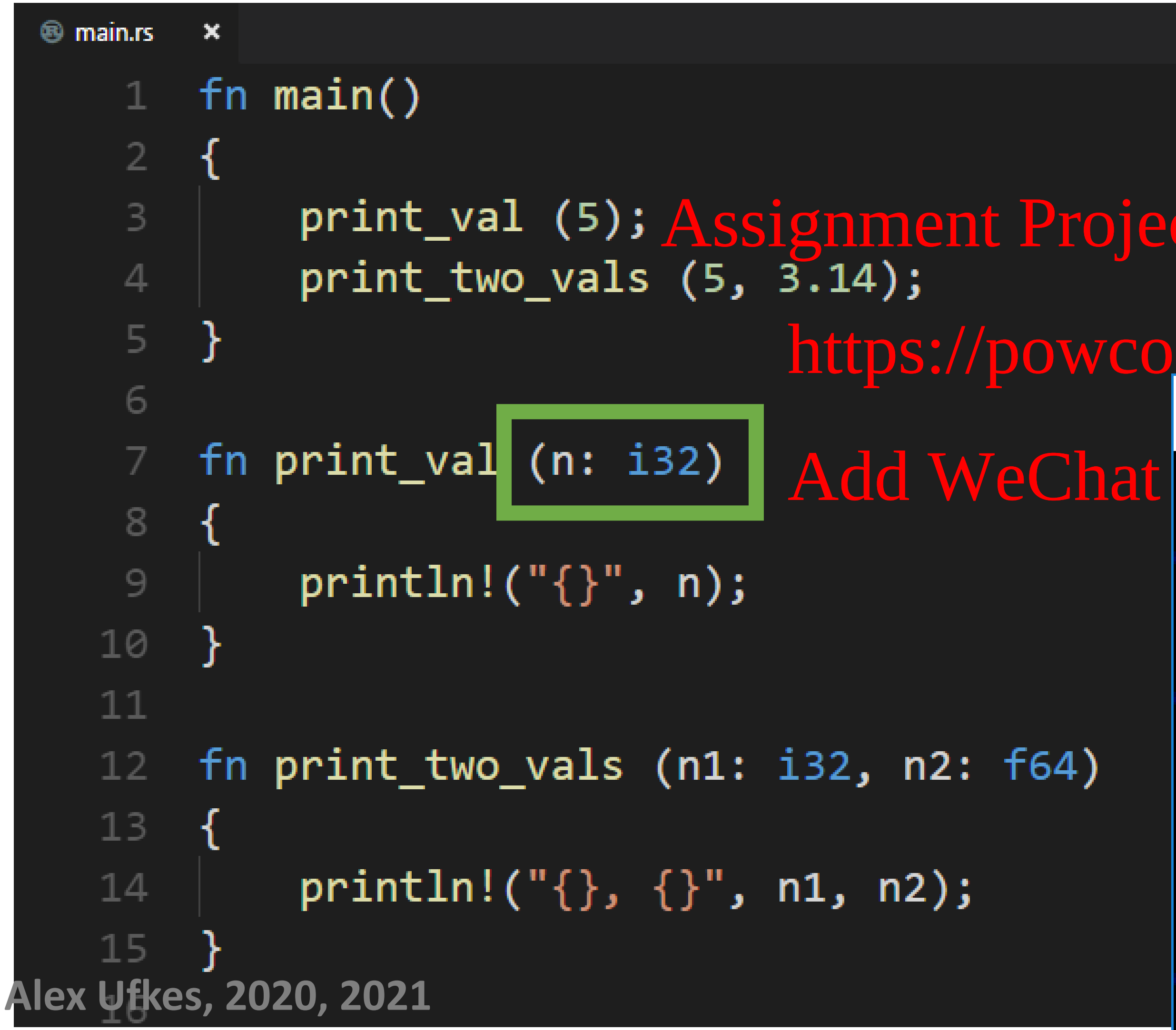
```rust
fn main()
{
    like_main_but_not_as_good ();
}

fn like_main_but_not_as_good ()
{
    println!("Hello World!");
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
Hello World!

C:\_RustCode>_
```

Unlike C/C++, Rust doesn’t care about ordering

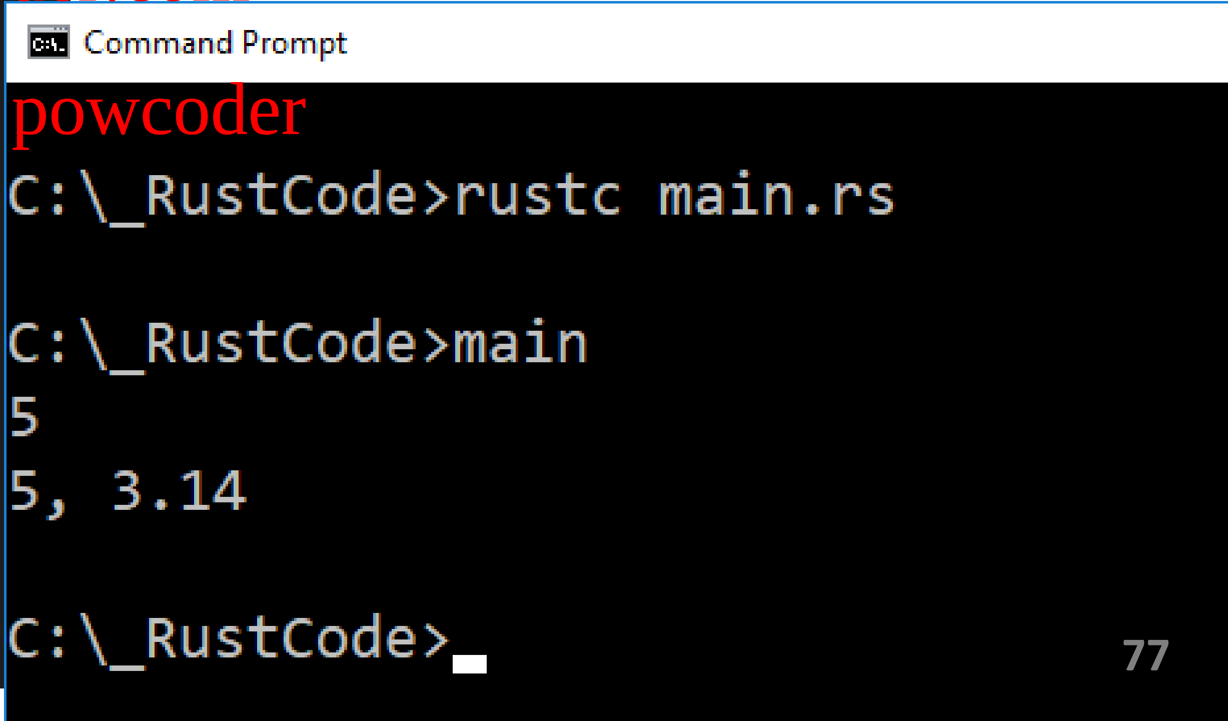
Parameters



```rust
fn main()
{
    print_val (5);
    print_two_vals (5, 3.14);
}

fn print_val (n: i32)
{
    println!("{}", n);
}

fn print_two_vals (n1: i32, n2: f64)
{
    println!("{}, {}", n1, n2);
}
```

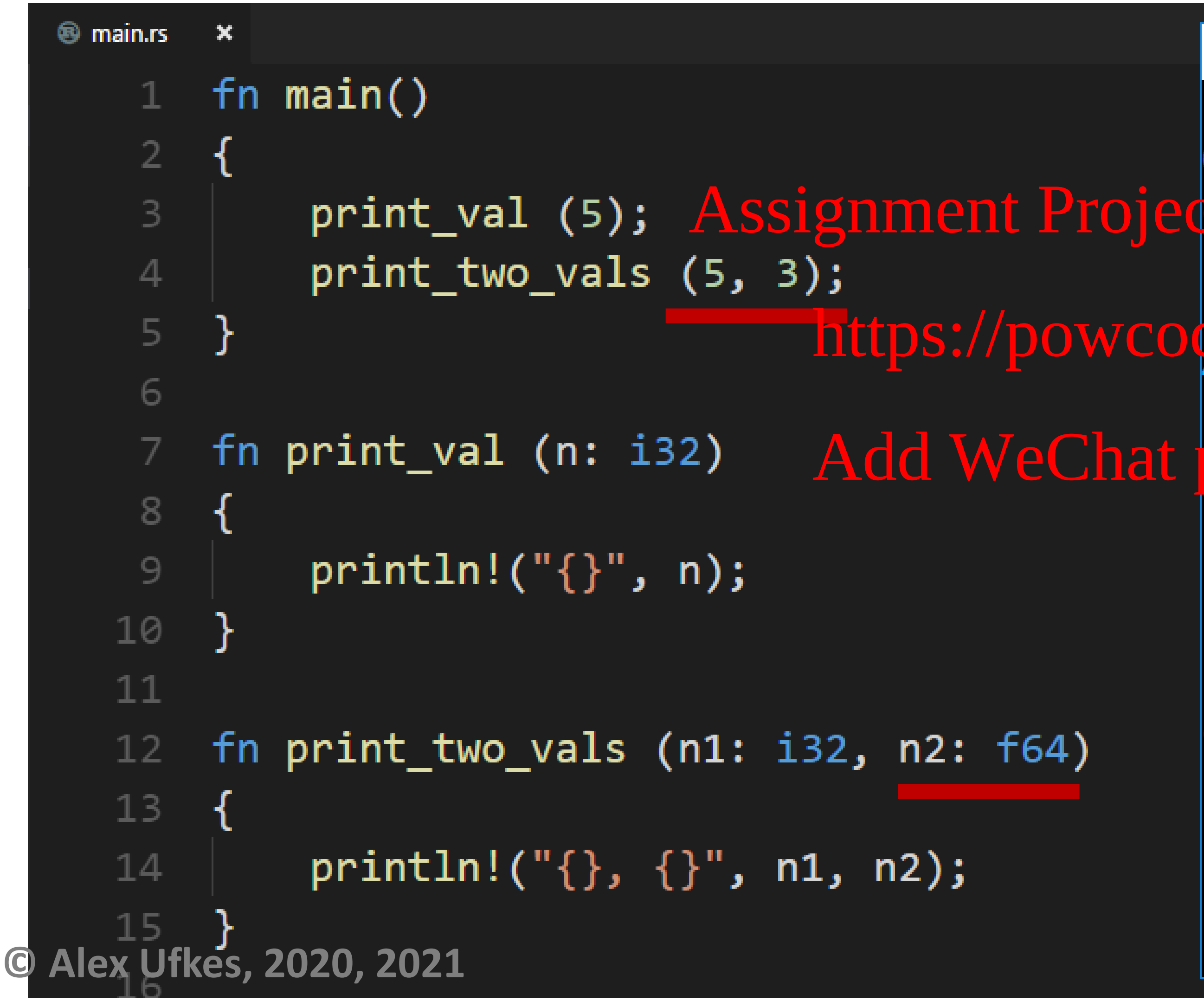
`identifier: type`

- Parameters separated by commas.
- Indicating type is ***mandatory***
- Nothing too unusual here

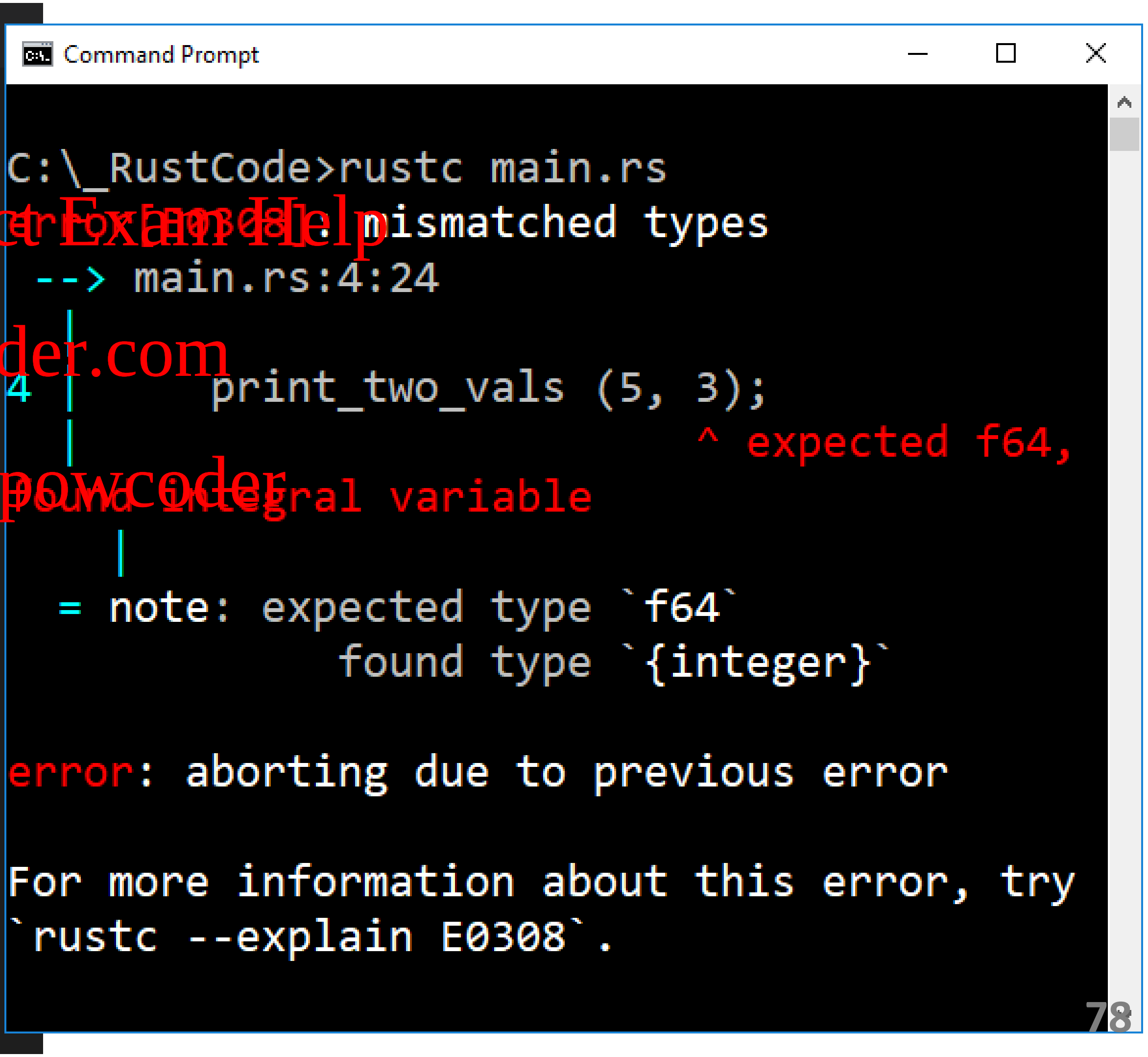


```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
5
5, 3.14

C:\_RustCode>_
```

Careful Now…



```
fn main()
{
    print_val (5);
    print_two_vals (5, 3);
}

fn print_val (n: i32)
{
    println!("{}", n);
}

fn print_two_vals (n1: i32, n2: f64)
{
    println!("{}, {}", n1, n2);
}
```



```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:4:24
  |
4 |     print_two_vals (5, 3);
  |                        ^ expected f64,
found integral variable
    |
  = note: expected type `f64`
             found type `{integer}`

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0308`.
```

Statements & Expressions

Rust is *primarily* expression based, but still has statements.

Two types of statements:

- Declaration statements return nothing
- Expression statements return empty tuple ()

```
let x = 6;        // This is a declaration statement
```

The above does not return a value. We can’t do the following:

```
let y = (let x = 6);
```

Statements & Expressions

Rust is *primarily* expression based, but still has statements.

Two types of statements:

- Declaration statements return nothing
- Expression statements return empty tuple `()`

```
5 + 2;      // This is an expression statement
```

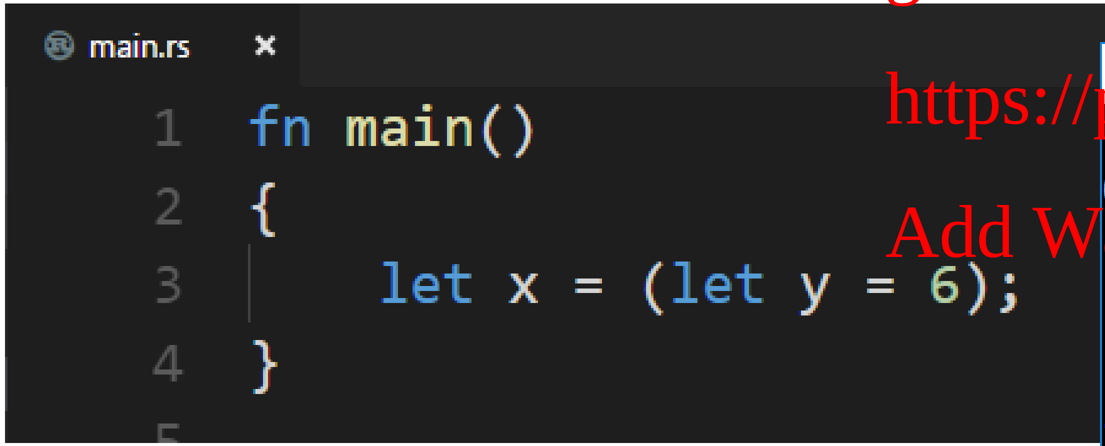
The above expression is evaluated, but the result is ignored (not saved).

`5 + 2` is an **expression**. It evaluates to 7.

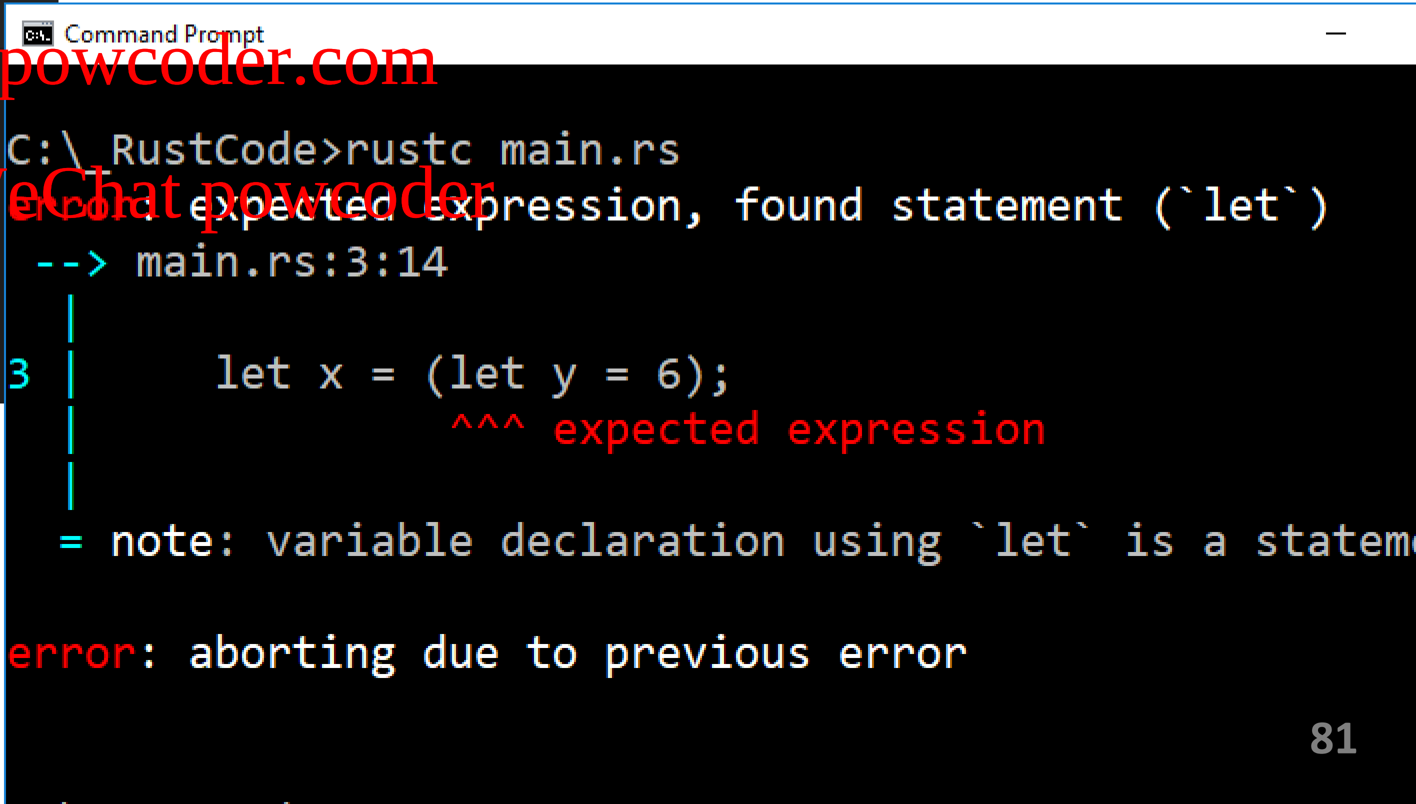
`y = 5+2;` is an **expression statement**. It returns (), but the result of the nested expression 5+2 is saved to y

Statements & Expressions

```
let y = (let x = 6);
```

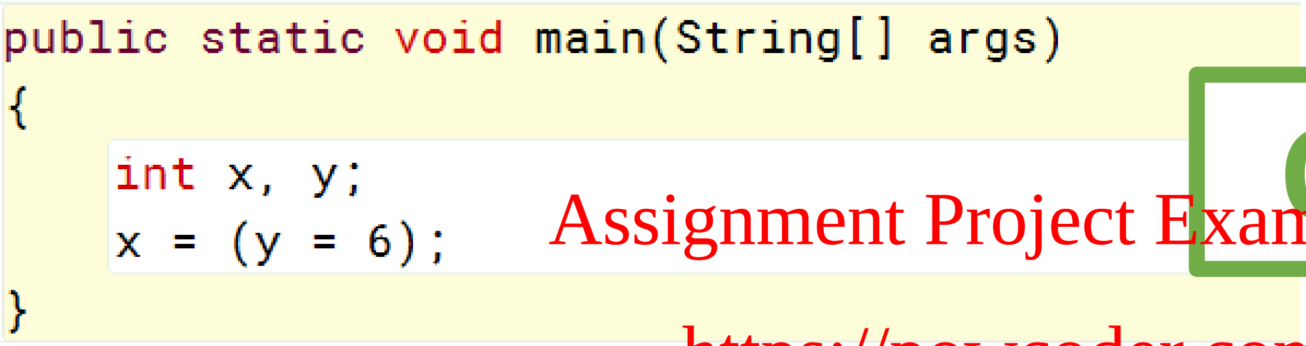


```
fn main()
{
    let x = (let y = 6);
}
```



```
C:\ RustCode>rustc main.rs
error: expected expression, found statement (`let`)
 --> main.rs:3:14
  |
3 |     let x = (let y = 6);
  |              ^^^ expected expression
  |
  = note: variable declaration using `let` is a stateme

error: aborting due to previous error
```

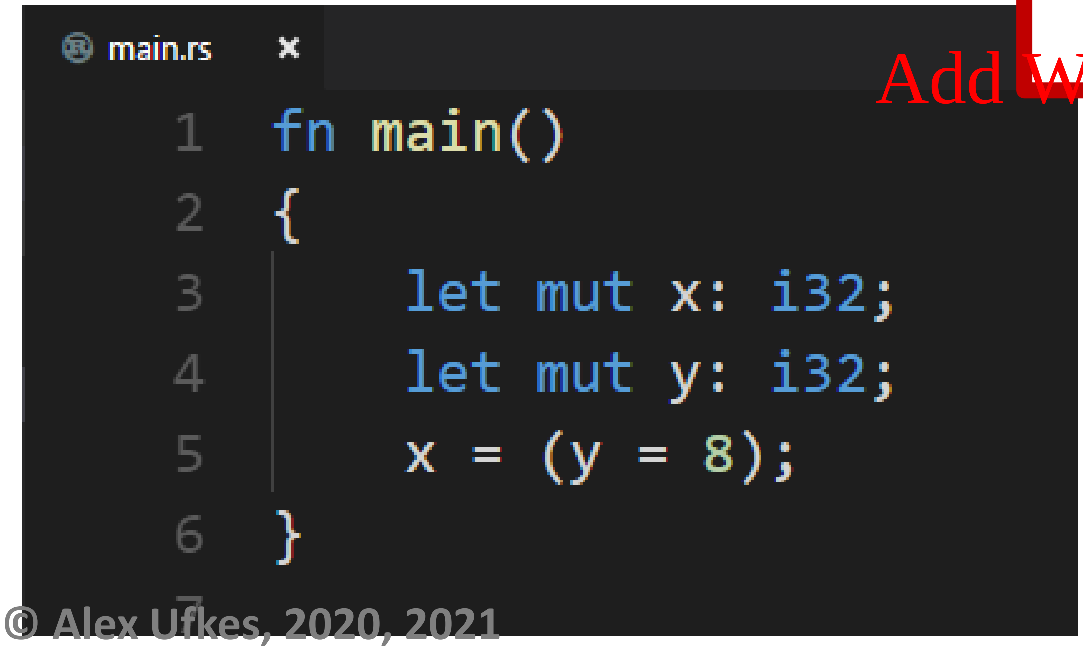
Statements & Expressions



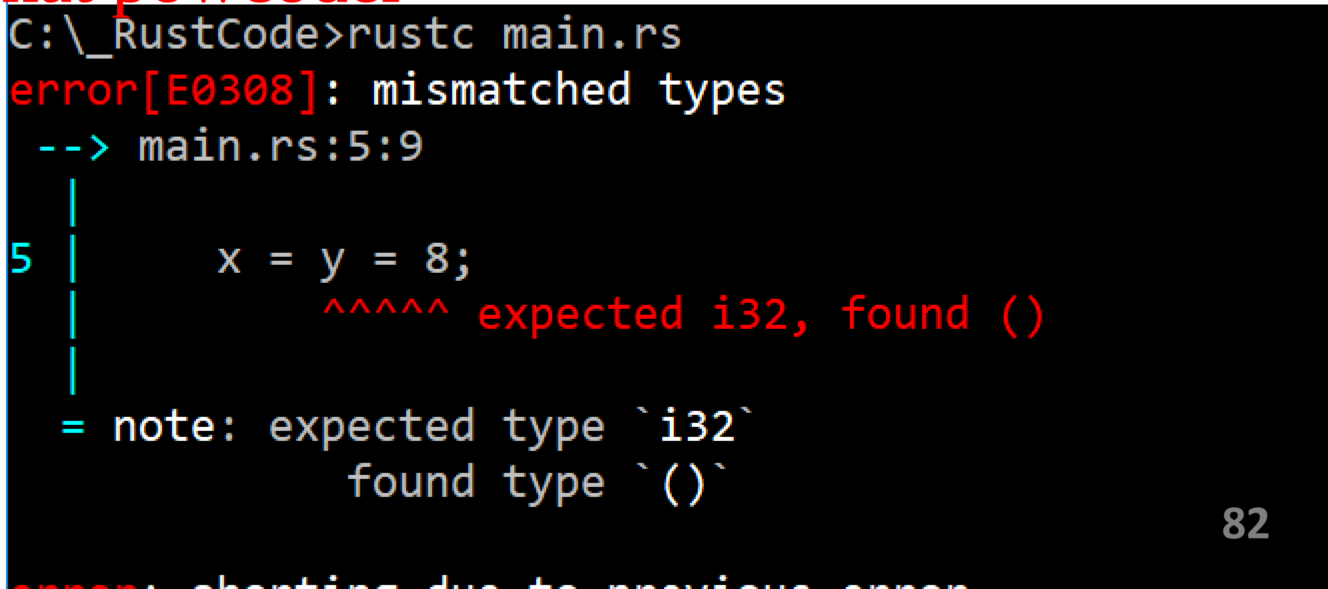
```
public static void main(String[] args)
{
    int x, y;
    x = (y = 6);
}
```

OK

Not OK… but what does this error mean?

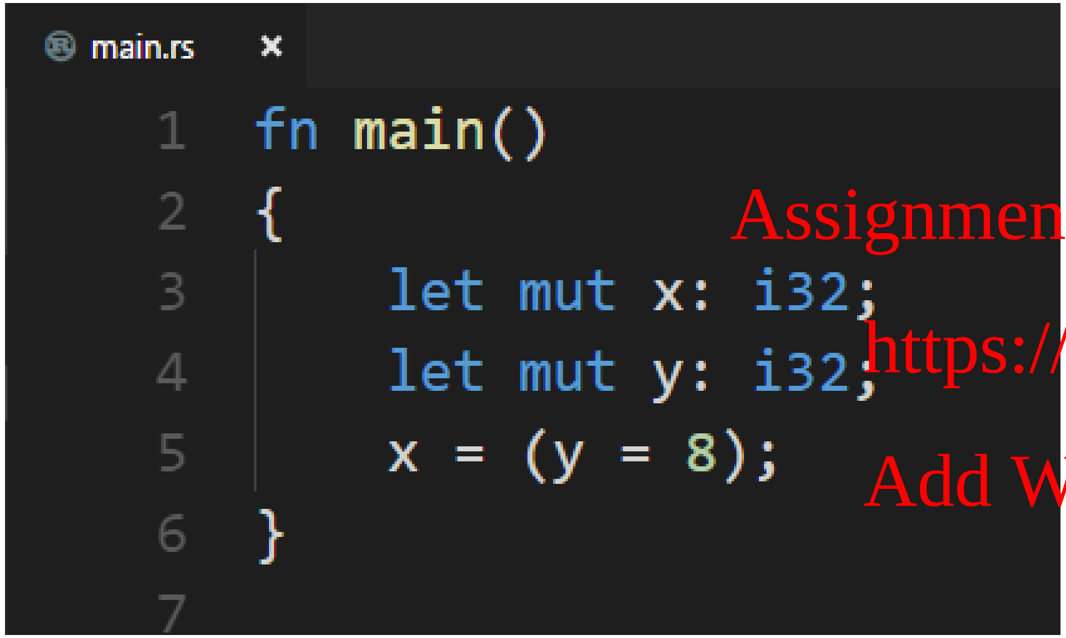


```
fn main()
{
    let mut x: i32;
    let mut y: i32;
    x = (y = 8);
}
```

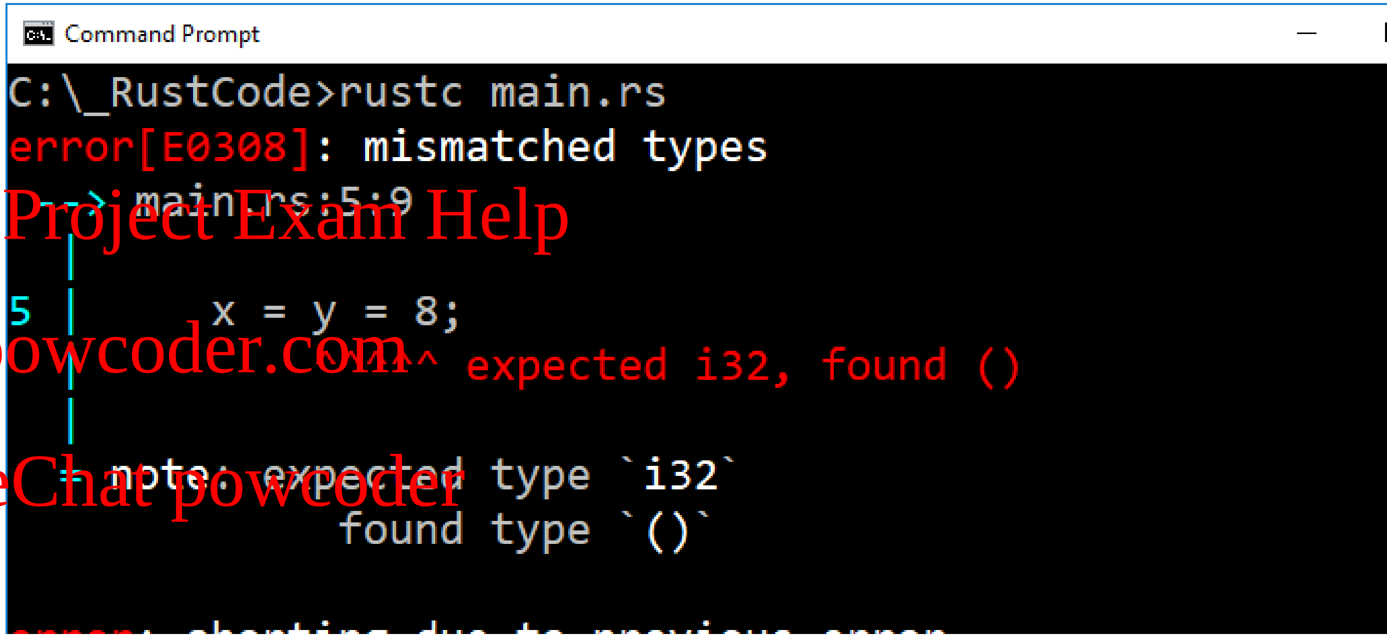


```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:5:9
  |
5 |     x = y = 8;
  |         ^^^^^ expected i32, found ()
  |
  = note: expected type `i32`
             found type `()`

error: aborting due to previous error
```

Statements & Expressions



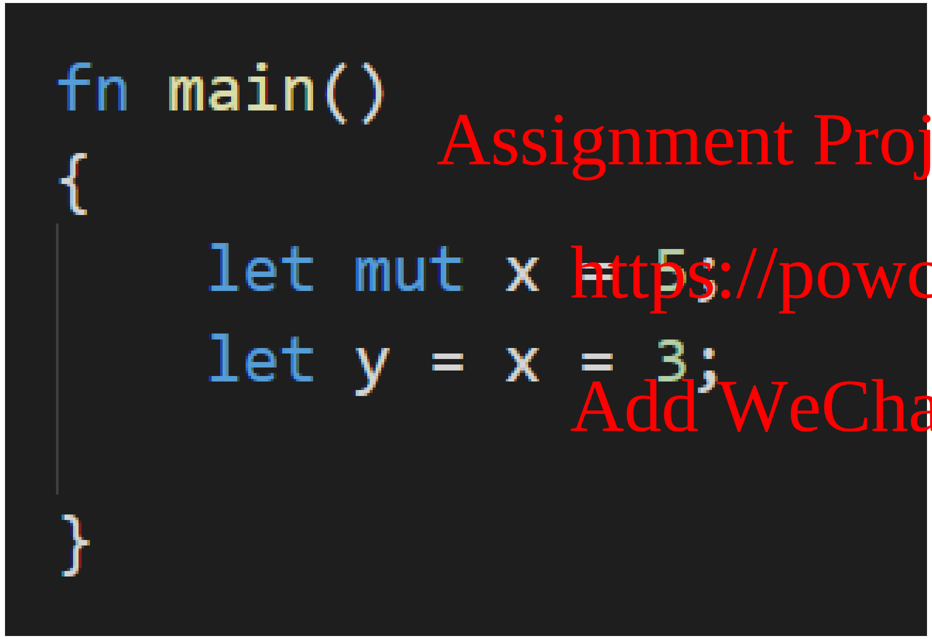
```rust
fn main()
{
    let mut x: i32;
    let mut y: i32;
    x = (y = 8);
}
```



```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:5:9
  |
5 |     x = y = 8;
  |         ^^^^^ expected i32, found ()
  |
  = note: expected type `i32`
             found type `()`

error: aborting due to previous error
```

- Variable **y** gets re-assigned.
- The *expression statement* **(y=8)** returns an empty tuple in Rust.
- Can't assign an empty tuple to a variable declared to hold **i32**!

Statements & Expressions



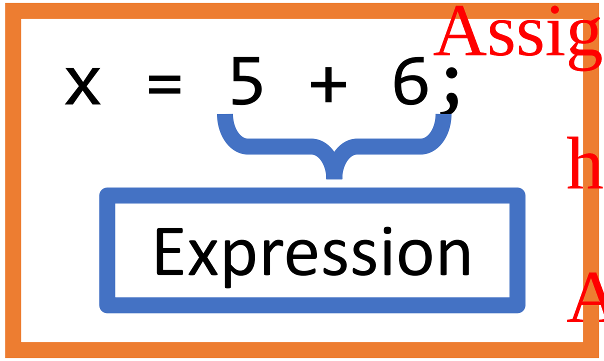
```
fn main()
{
    let mut x = 5;
    let y = x = 3;

}
```

Here:

- Value of x will be 3
- Value of y will be () empty tuple

Statements & Expressions

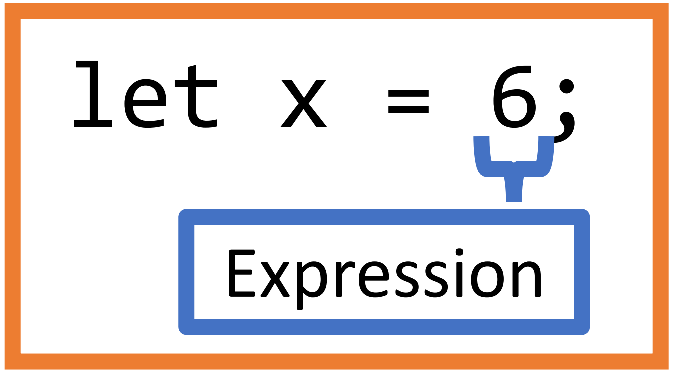
```
x + 6               // This is an expression

x = 5 + 6;          // This is an expression statement
                    // containing an expression
```



Expression statement

Declaration statement

In fact:

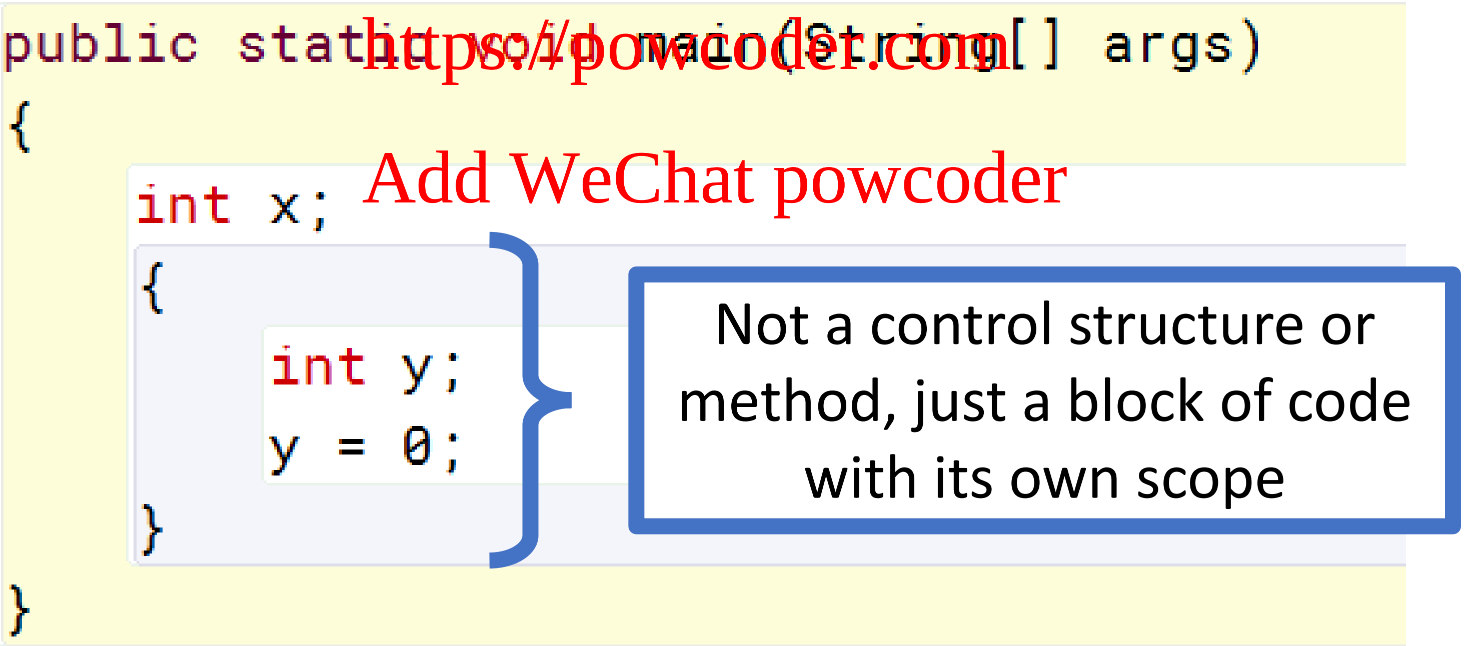


```
let x = 6;
```

Scope Blocks as Expressions

Creating a new scope block?

We can do this in Java and C/C++, though again it isn't so common:

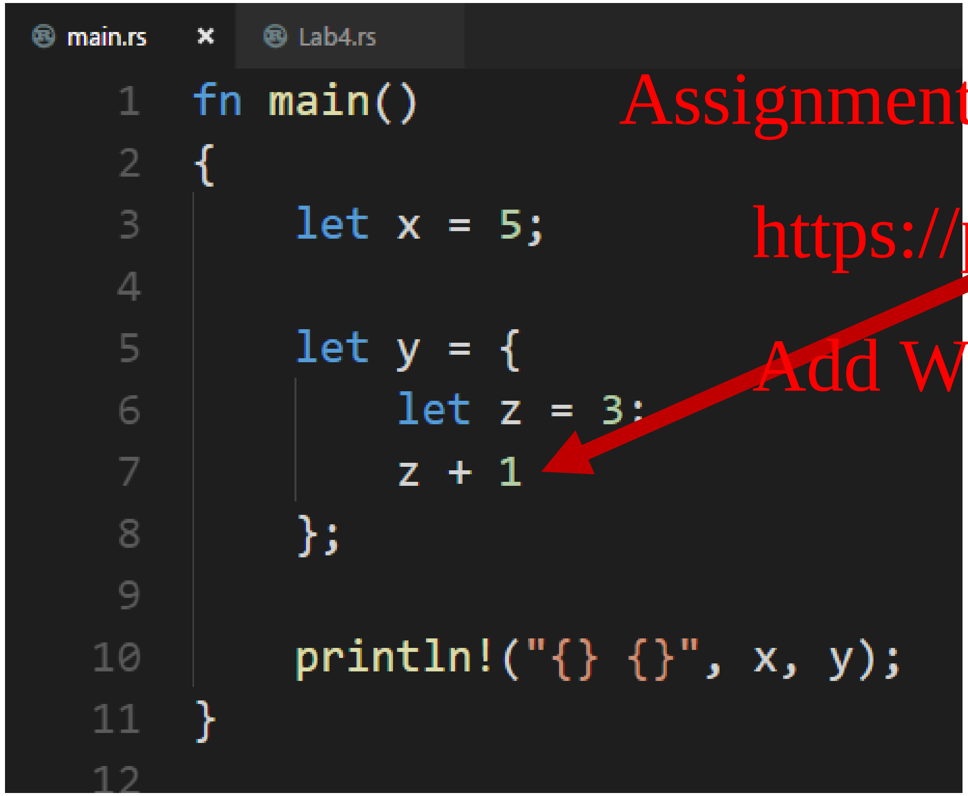


```
public static void main(String[] args)
{
    int x;
    {
        int y;
        y = 0;
    }
}
```

Not a control structure or method, just a block of code with its own scope

Scope Blocks as Expressions

Scope blocks like this are expressions in Rust:



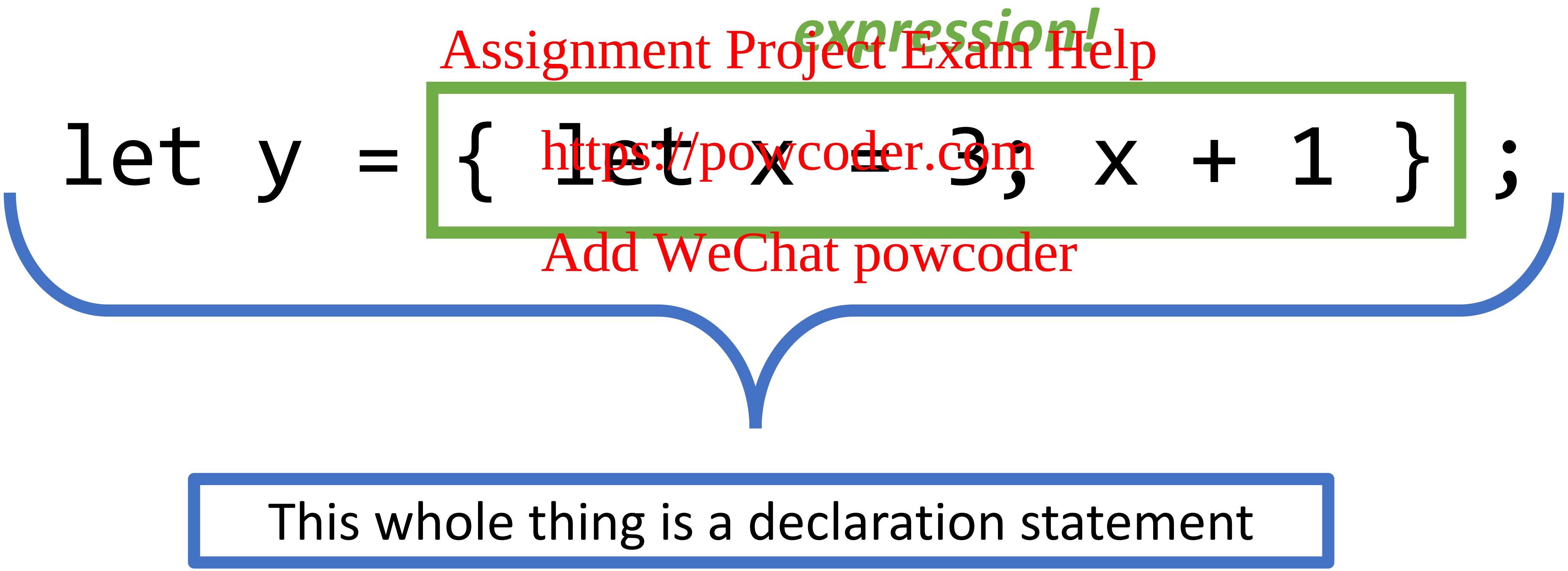
```
fn main()
{
    let x = 5;

    let y = {
        let z = 3;
        z + 1
    };

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

There's a few things going on here:

- We're trying to bind a value to y.
- Thus, the block { } should evaluate to something.
- Notice there's no semicolon after z + 1
- z + 1 is an expression.
- Adding a semi-colon would make it an *expression statement*.
- Thus, the block { } would return ( ).
- Probably not what we want.

Scope Blocks as Expressions

Scope blocks like this are expressions in Rust:



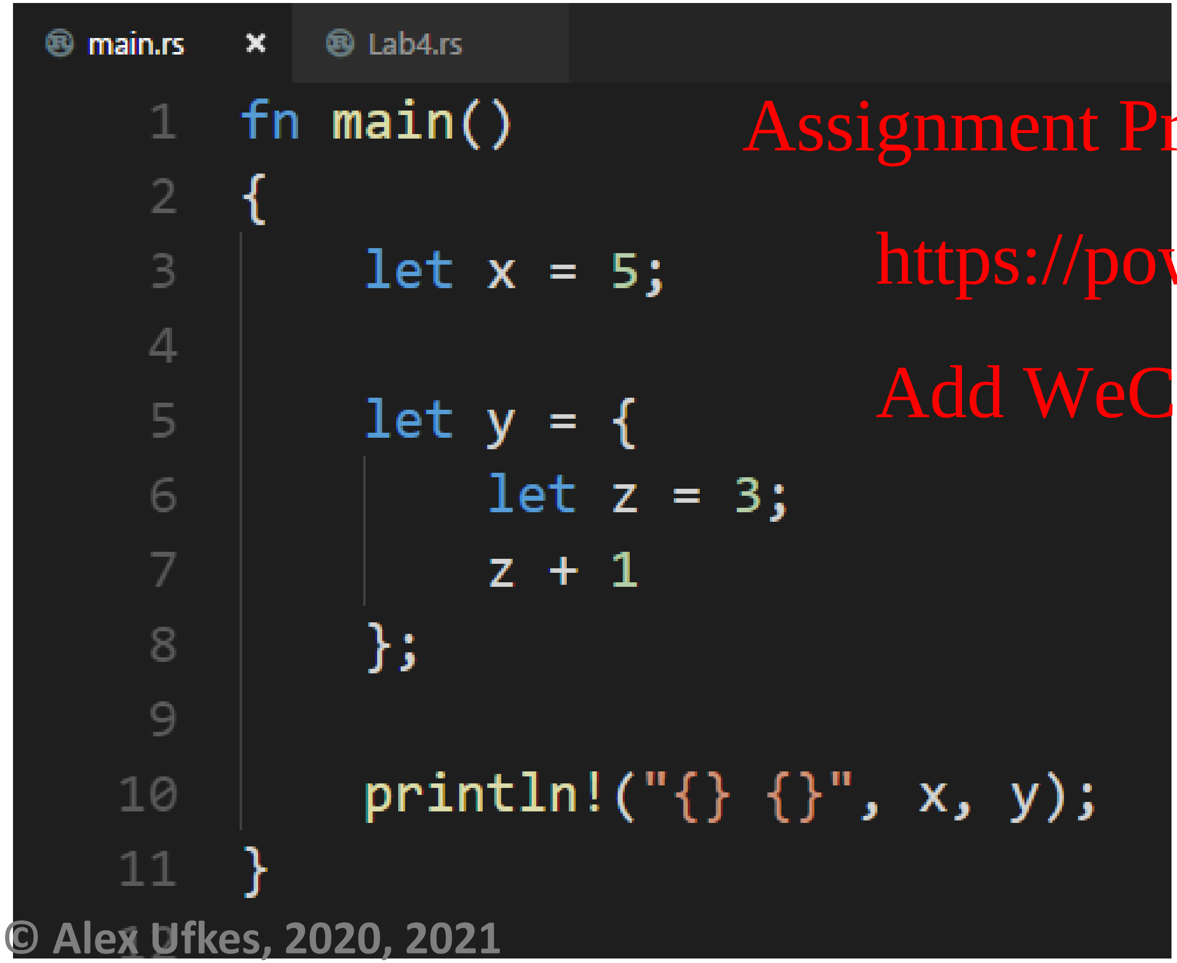
expression!

```
let y = { let x = 3; x + 1 };
```

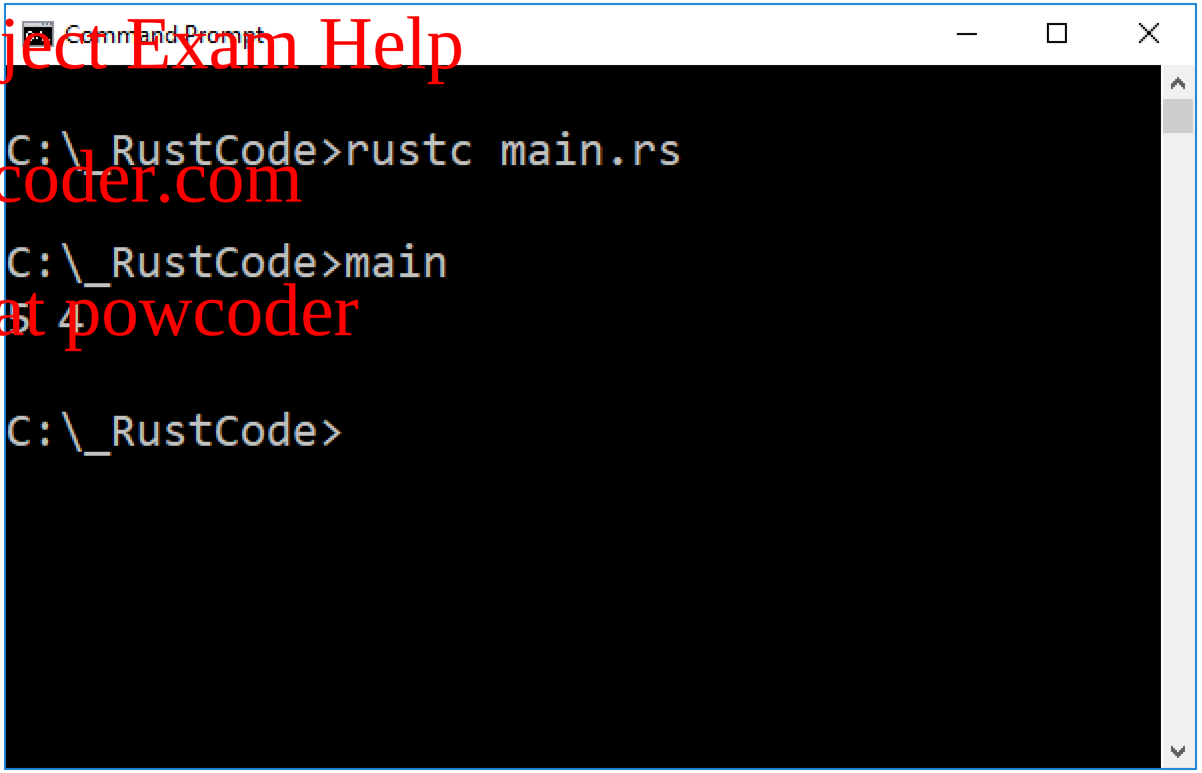
This whole thing is a declaration statement

Scope Blocks as Expressions

Scope blocks like this are expressions in Rust:



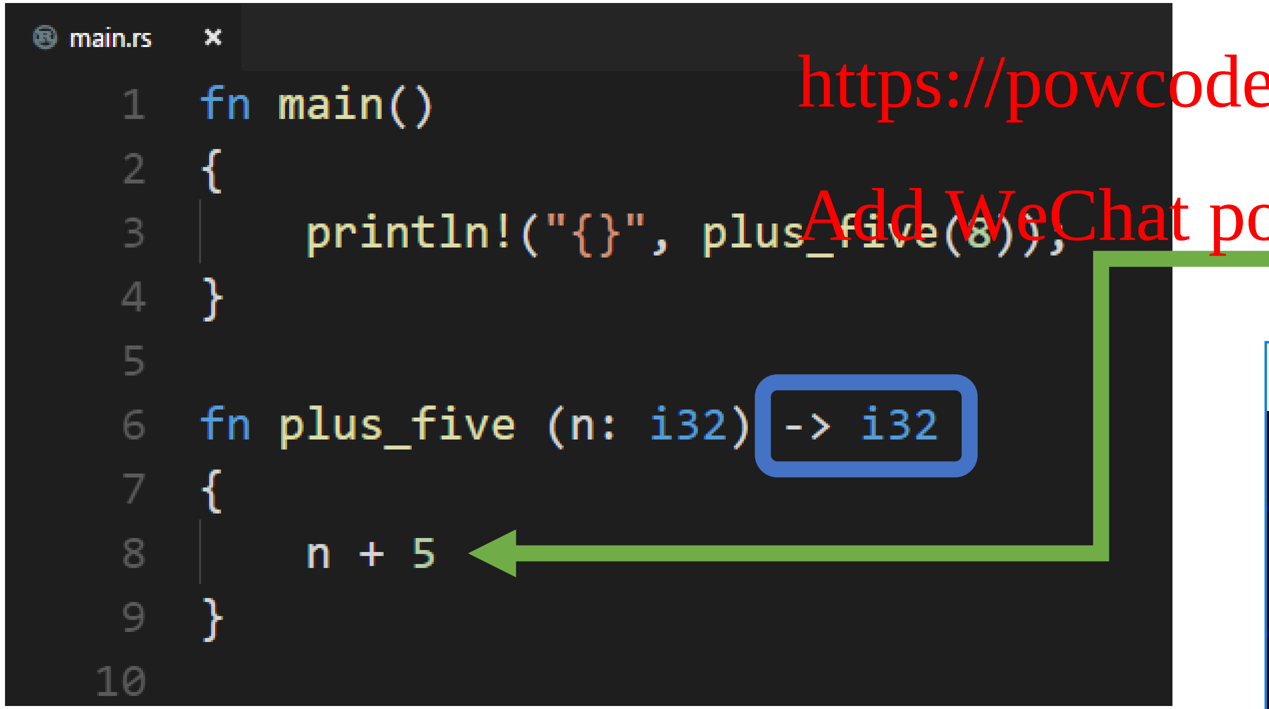
```rust
fn main()
{
    let x = 5;

    let y = {
        let z = 3;
        z + 1
    };

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



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
5 4

C:\_RustCode>
```

Return Value

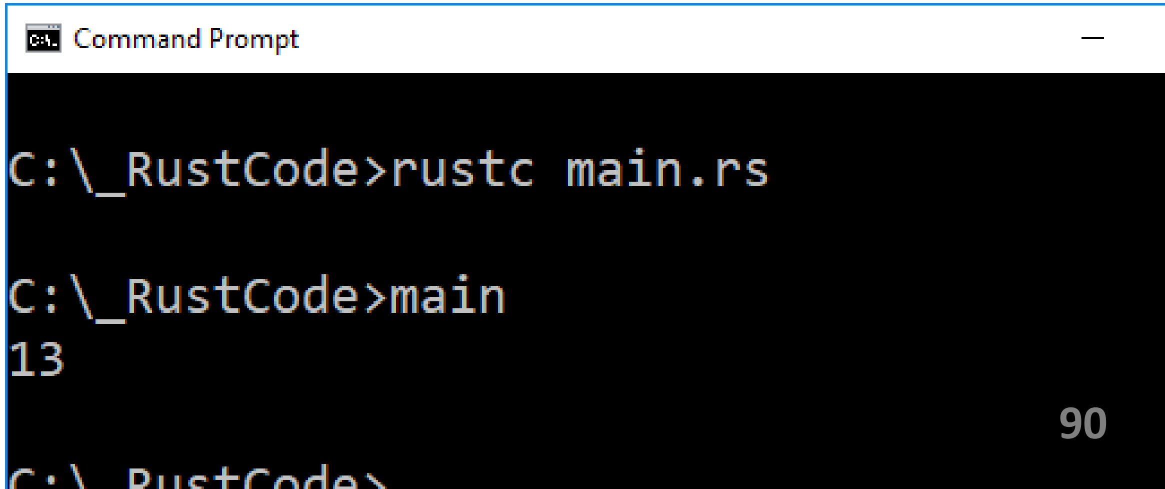
Think of functions the same way.
The last line should be an expression – no semi-colon.



```rust
fn main()
{
    println!("{}", plus_five(8));
}

fn plus_five (n: i32) -> i32
{
    n + 5
}
```

-> type

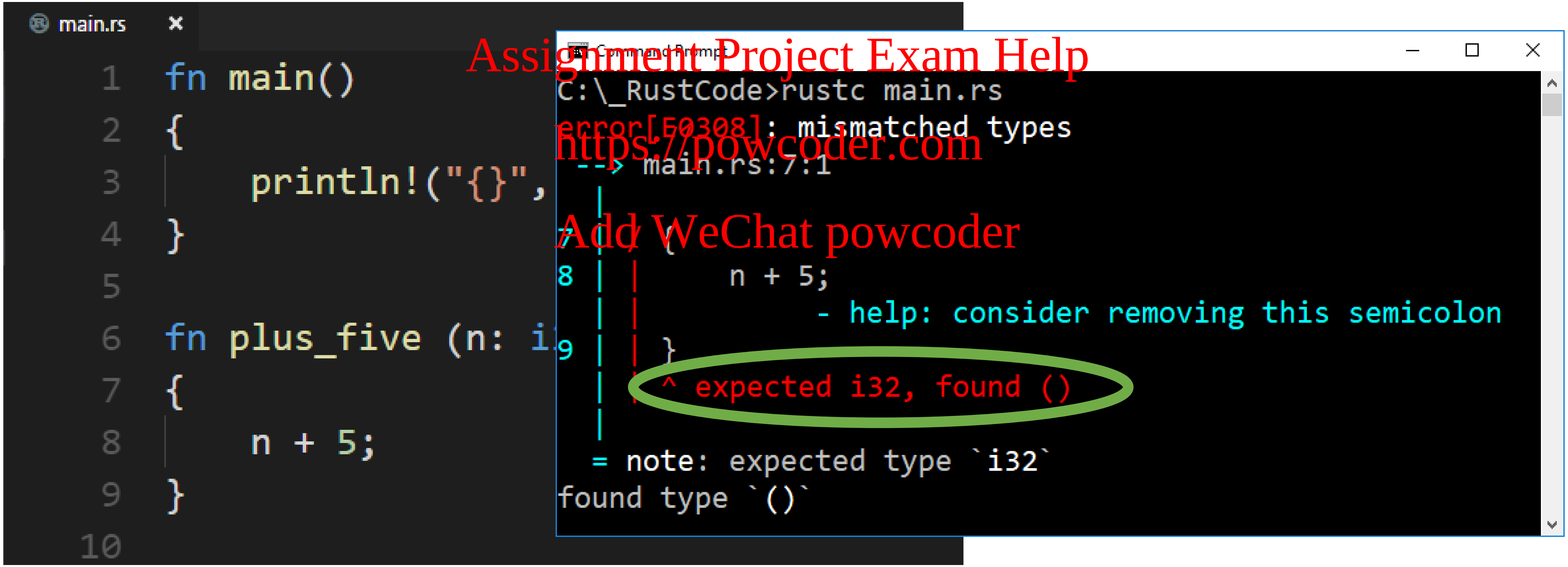
- Explicitly indicate return type
- Result of expression gets returned



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
13

C:\_RustCode>
```

Return Value

Add semicolon? It becomes expression statement, returns ( ), type mismatch:



```rust
fn main()
{
    println!("{}",
}

fn plus_five (n: i
{
    n + 5;
}
```

```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:7:1
  |
7 | / {
8 | |     n + 5;
  | |          - help: consider removing this semicolon
9 | | }
  | |_^ expected i32, found ()
  |
  = note: expected type `i32`
found type `()`
```

Fantastic Rust Reference:

https://doc.rust-lang.org/book/second-edition/