C/CPS 506

Compara **Languages**

Prof. Ale

Topic 9: Rust intro & typing

Ryerson University

Notice!

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

The copyright to this ex Ufkes. Students registered in course CCPS 506 can us al 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 language.
- Don’t forget about the assignments!

Moving on…

erative.

Rust is an imperative language. How 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
  soring the project in 2009
  d in 2010
- Rust com fully tested in 2011
- Pre-alpha ased 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

% using who want to keep using

Also #1 in 2017 and 2016!

In Industry?

Mozilla in collaboration with Samsung

- Parallel web browser engine

Dropbox

- Magic Pocket file system, peta machines

Tor (The Onion Router)

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

Rust Features

Systems Programming Language:

- In contrast with *application* nguages.
- includes things like operat , utility software, device pilers, linkers, etc.
- System languages tend to feature more direct access to physical hardware of a given machine.

Rust Features

Syntax:

- Similar to C/C++
- elimited by { }
  structures supported
  (if, e , for, etc.)
- Suppo atching! (match)
- Need not use return, last expression creates return value
- Functions largely composed of expressions

Rust Features

<u>Memory Safety:</u>

- Rust is designed to be *memory safe*

pointers are not permitted.

“Null or dangling pointers are not permitted”

- In C, we’re allowed to ***try*** and access any memory we want.
- This code compiles!
  - Produces a run-time error when 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”

- 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:

to be *memory safe*

pointers are not permitted.

- What lists? Null pointers are useful.

- Rust d *tion* type, which can be used to test if a pointer has *Some* value or *None*
  - What does this remind you of?

Rust Features

Memory Management:

- Rust does not do garbage collection

ition is initialization

in C++

- Constr to acquire and initialize objects
- Resou *tion* 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 [illegible] by Haskell’s type class
- Suppor[illegible]rence for variables declared with `let` keyword.
- Compile error if inference fails.
- Keyword `mut` for mutable variables.

Rust Features

Pattern Matching:

ttern matching!

g is considered a

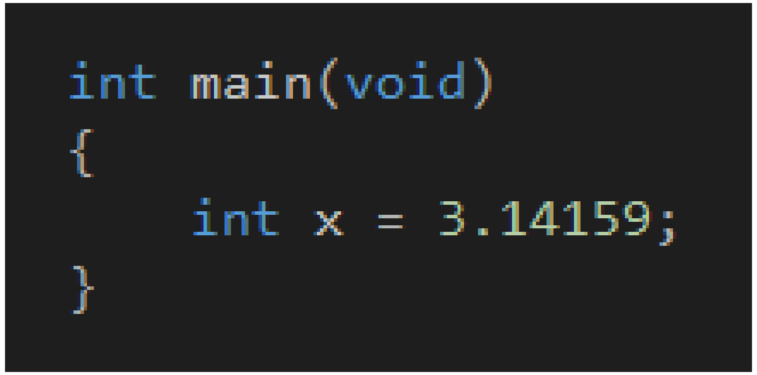
stickin eople learning Rust.

- We alr xperience with it

Rust & Safety

Strongly, statically typed

- Strong typing means limited implicit type
compile time.
nvert between nume
types e. 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;
}
```

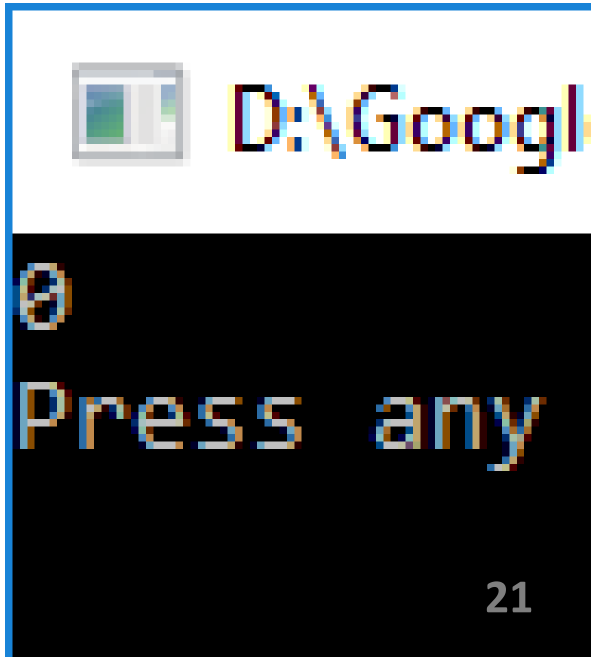
Java

Rust & Safety

No "Undefined Behavior"

- Null pointer dereferencing

dereference address 0



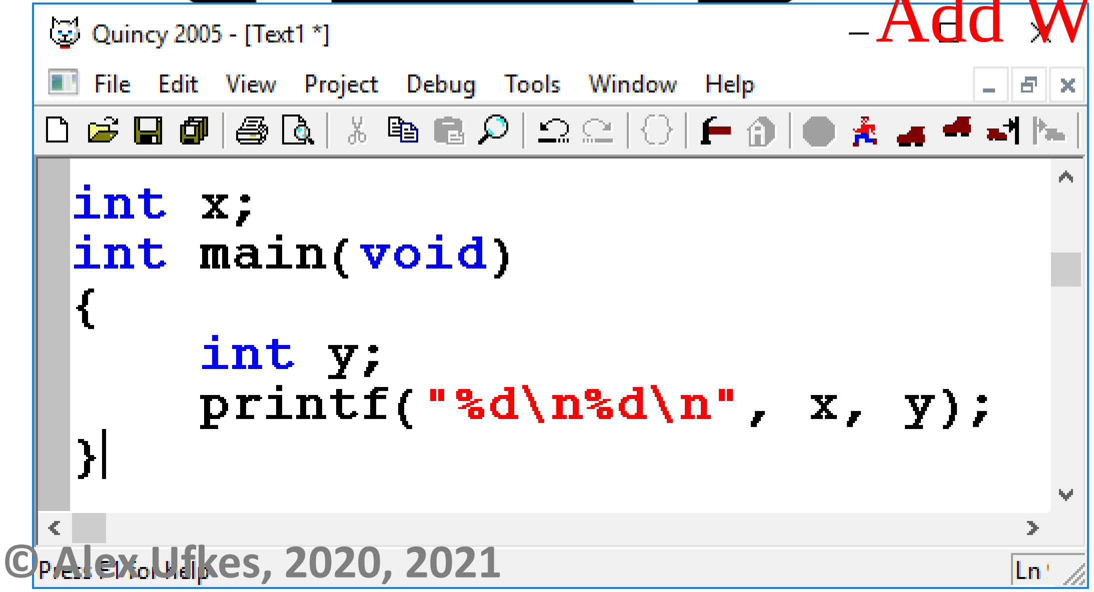
D:\Googl

```
0
Press any
```

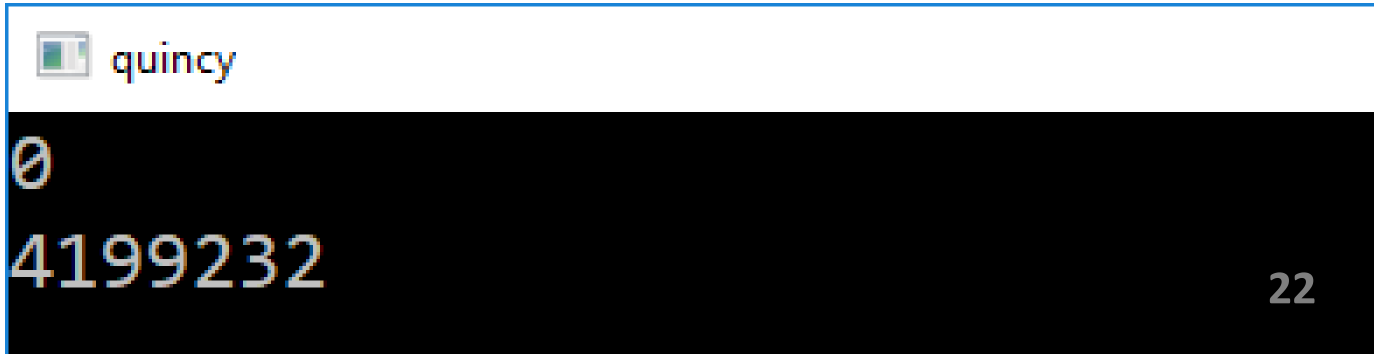
Rust & Safety

No “Undefined Behavior”

- Null pointer dereferencing
  dereference address 0
  before it’s initialized
  - In hatever was in
    In re 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

dereference address 0

before it's initialized

  - In hatever was in

    in re 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

Rust & Safety

No "Undefined Behavior"

- Signed integer overflow & optimization

X

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

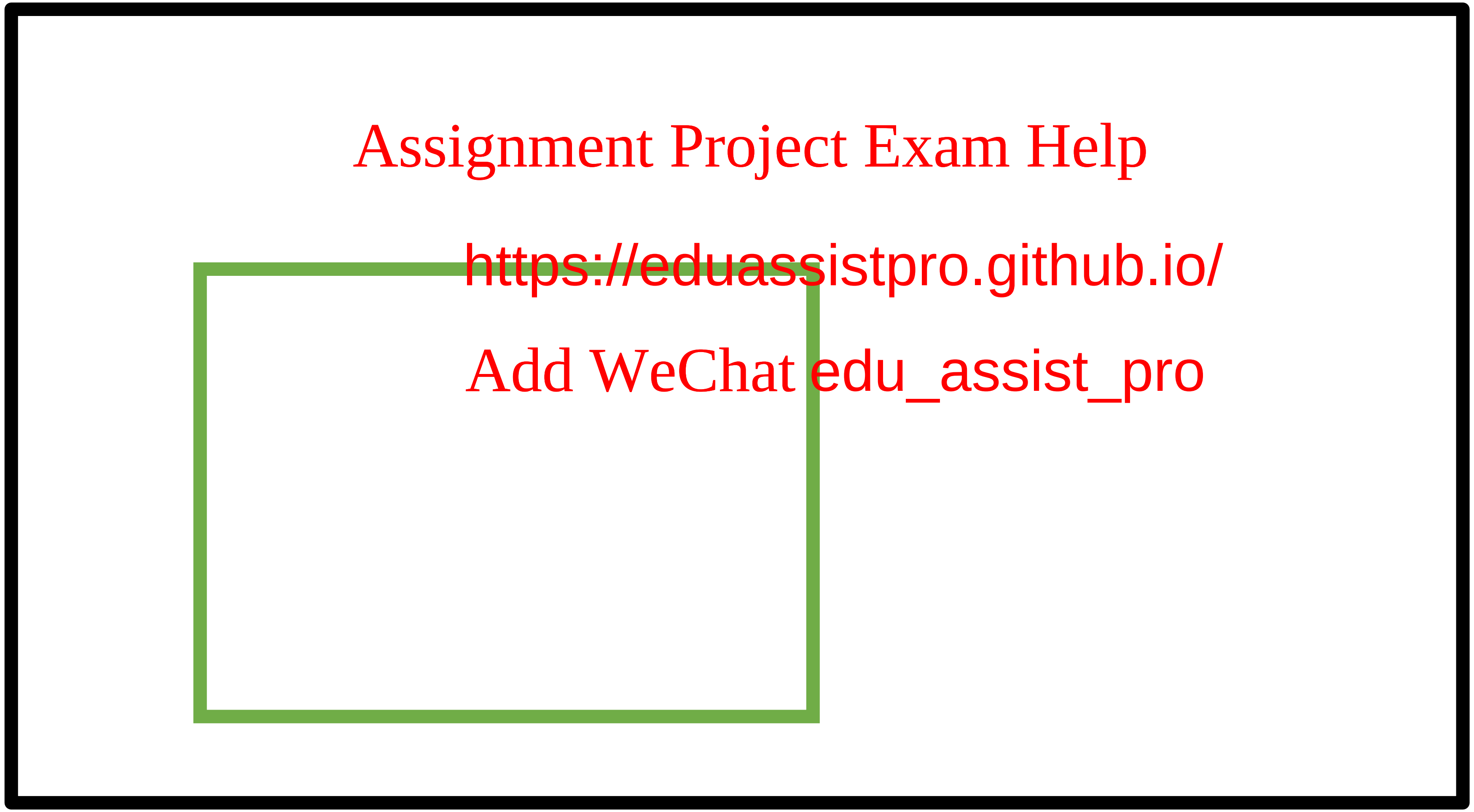
Rust Non-Goals

- 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. ubordinate goals.
- We do not i ete feature-set of C++, or any other language. Rust sh e majority-case features.
- We do not intend to be 100 0% 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

Installing Rust



Editing Rust Code

Any text editor will do, but I like VSCode:

Visual Studio Code:

- Supports Rust syntax coloring
- Useful for other languages

Compiling Rust Code

Command Line - **`rustc`**

https://www.rustaceans.org/

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

```
fn main() {
    print                    world!");
}
```

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

```
fn main() {
    print                          world!");
}
```

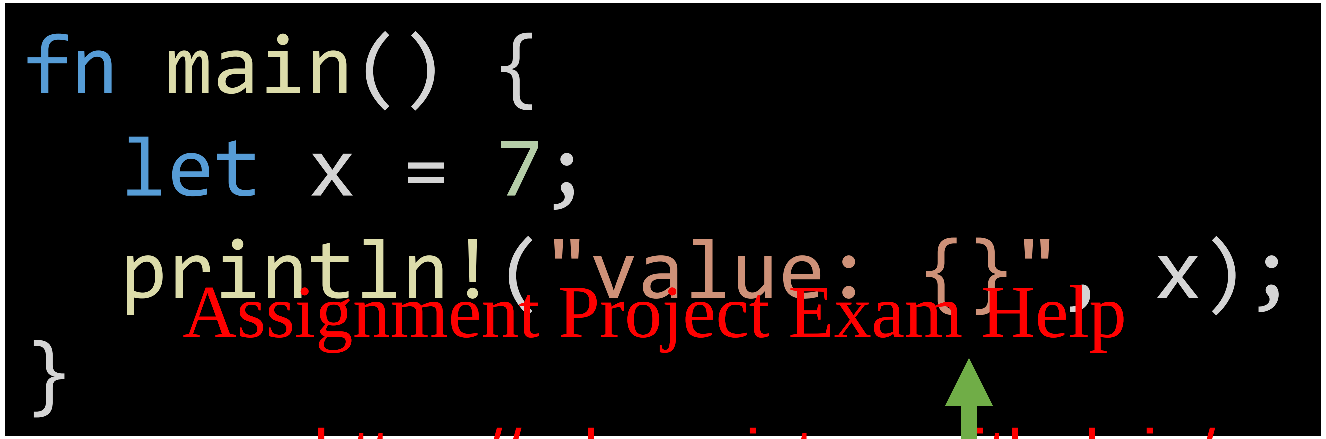
`println` vs `println!`

- 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

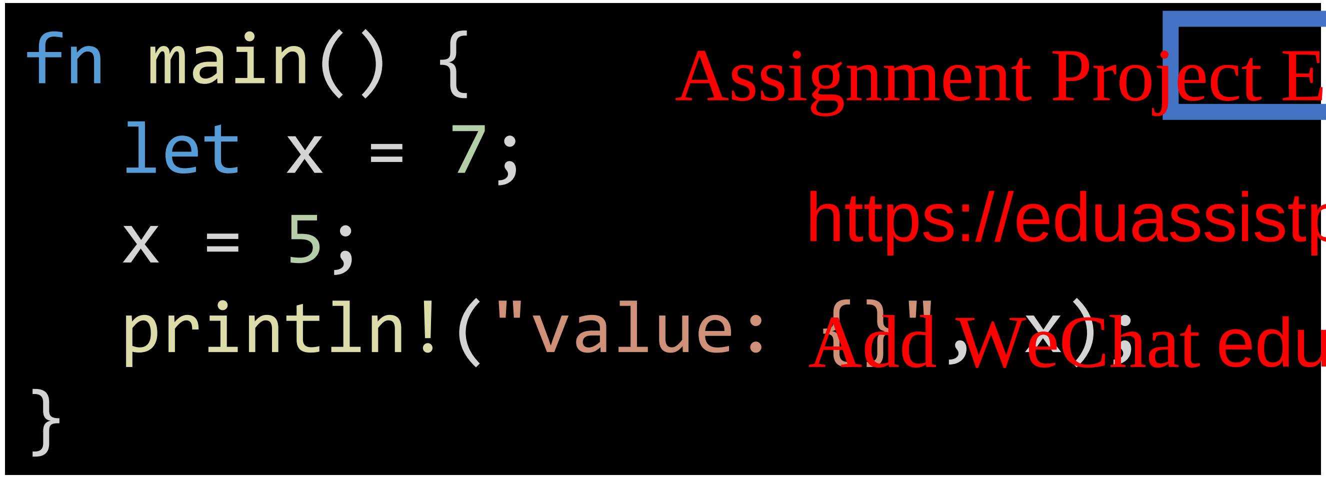
```
fn main() {
  let x = 7;
  println!("value: {}", x);
}
```



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

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

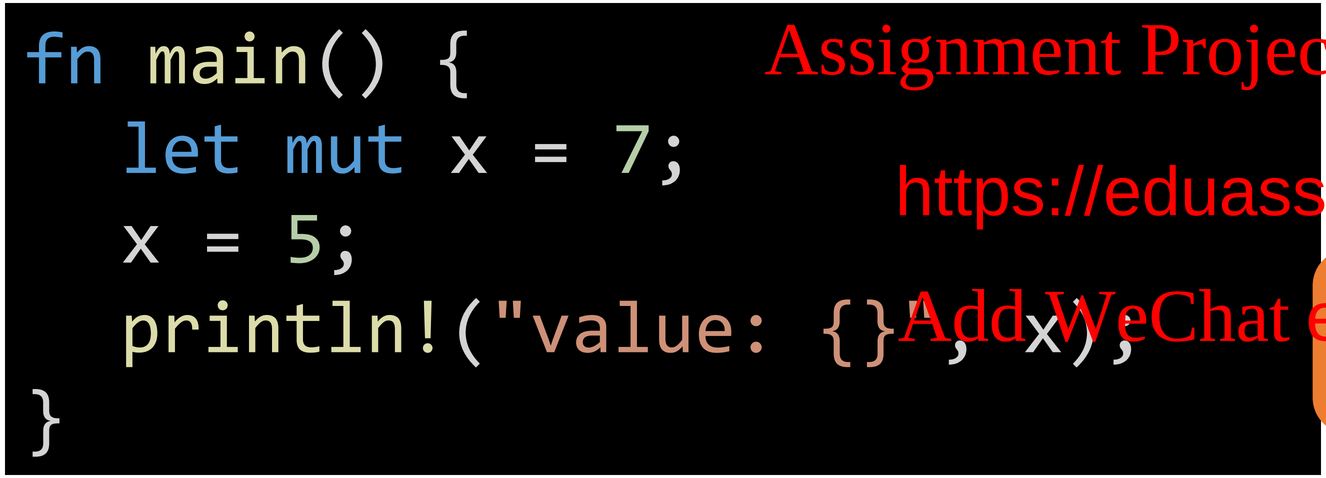
Variables



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

Mutable Variables

Use **mut** keyword:



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

- 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:

const instead of let

ys immutable

e declared in global

, unlike let

- Must indicate data type (u32)
- More on types coming up.

Constant/Global Variables

Can be declared in global scope, unlike `let`

Shadowing

Variables with the same name?

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

```
int r = 10;
                   {
                      = Math.sqrt(x);
}
```

BAD

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

OK

Shadowing

Variables with the same name?

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

```
int r = 10;
                 {
                r = sqrt(4.0);
}
```

OK

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

BAD

Shadowing

Variables with the same name?

Shadowing

Variables with the same name?

- What we're doing here is like binding in Haskell or Elixir. 't work with ble 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`?

- 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`?

- 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 na place, thus the **type mu**

- 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 repr

- Rust has four: integers, floatin leans, characters.

Compound types group multiple values:

- Two primitive compound types: tuples and arrays.

Scalar Types: Integers

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

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

Specify Type?

Rust has type inference, but we can be explicit:

Integer Literals

In addition to just writing the value…

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

Bytes can be character literals

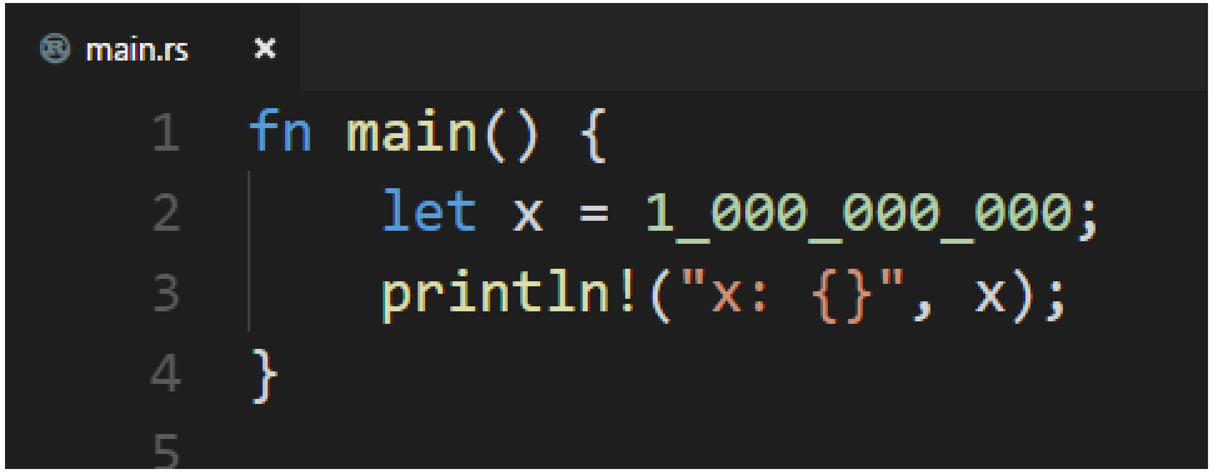
Notice the _

dy visual sugar

nt the zeroes in 1000000000.

is this?

- E 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

Default

Numeric Operations

Numeric Operations

Mixed Expressions?

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

Mixed Expressions?

Mixed Expressions?

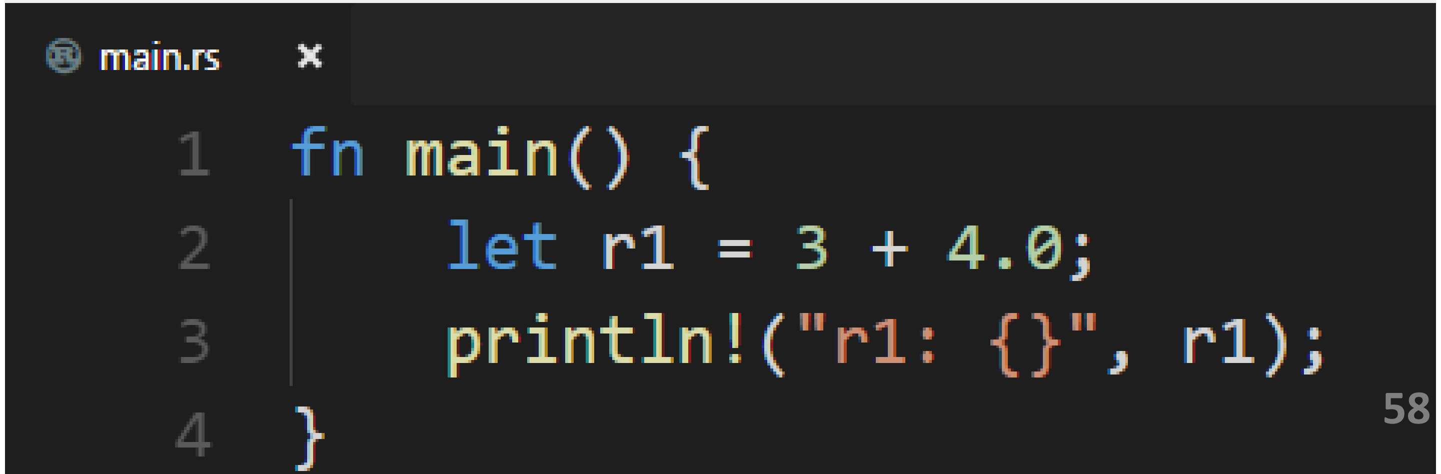
Cast using: **as** ***type***

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

Finally!

Mixed Expressions?

Division may truncate, good reason to avoid implicit conversion…



```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 an sing different instruct s on CPU.
- It's possible to introduc recision!
- An integer in binary is *e* *e*.
- 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.

IEEE-754

Scalar Types: Boolean

`true`, `false`. Easy:

Scalar Types: Characters

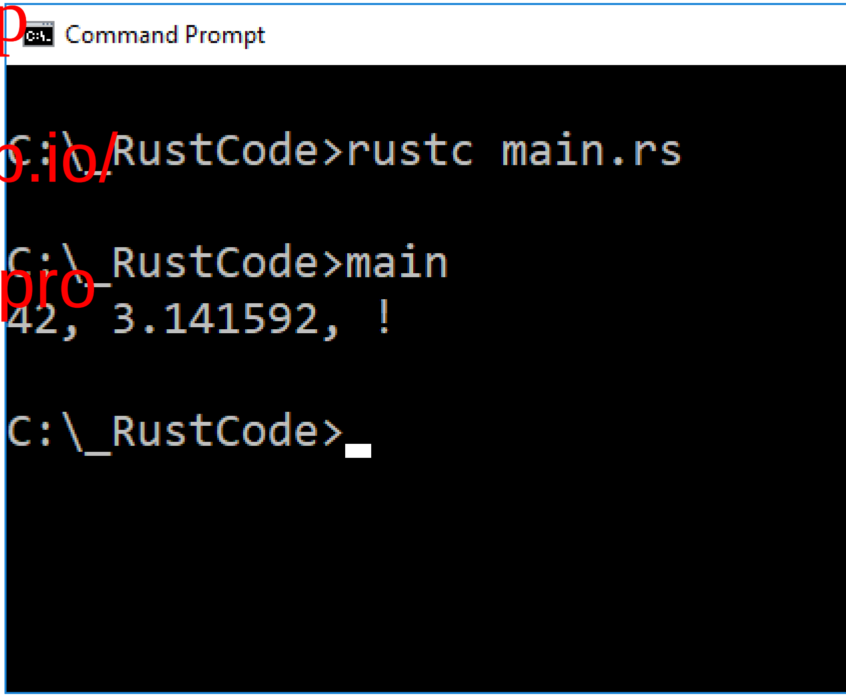
Rust supports Unicode:

Compound Types: Tuples

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

Accessing Elements

De-structuring!

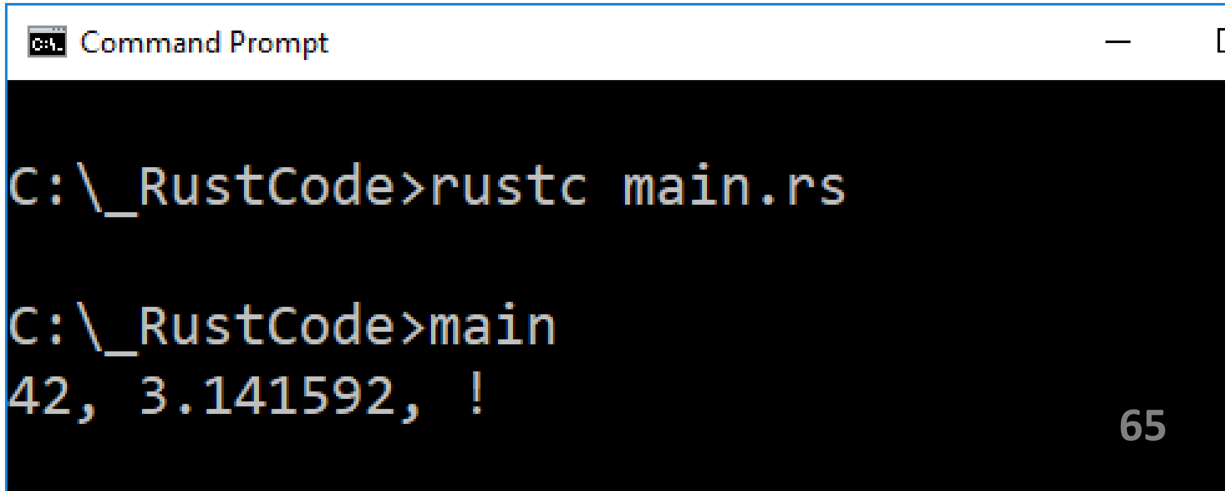


```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
42, 3.141592, !

C:\_RustCode>_
```

Accessing Elements

Can also access directly:

Can we go out of bounds?



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

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

Accessing Elements

Out of bounds:

Compile error in Rust

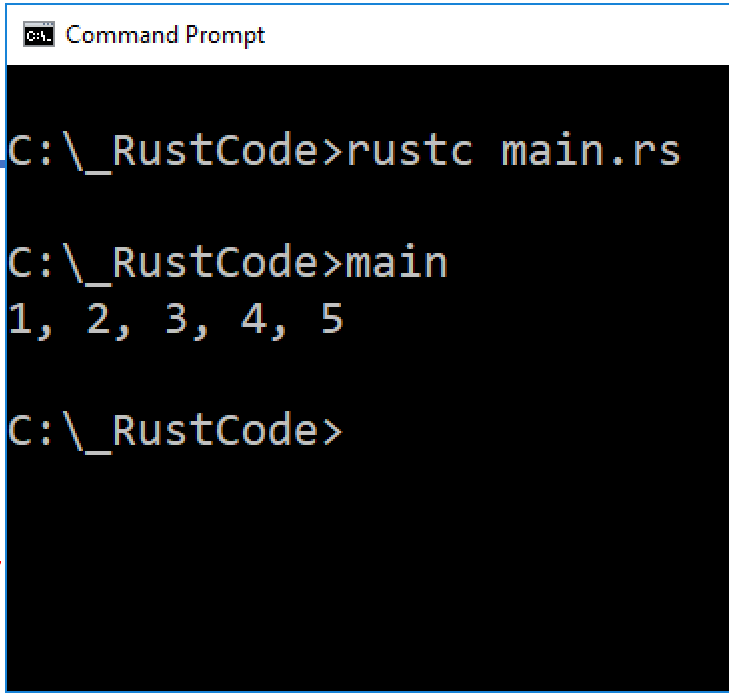
Accessing Elements

Can we fool it?

Nope.

Compound Types

Arrays



```
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.

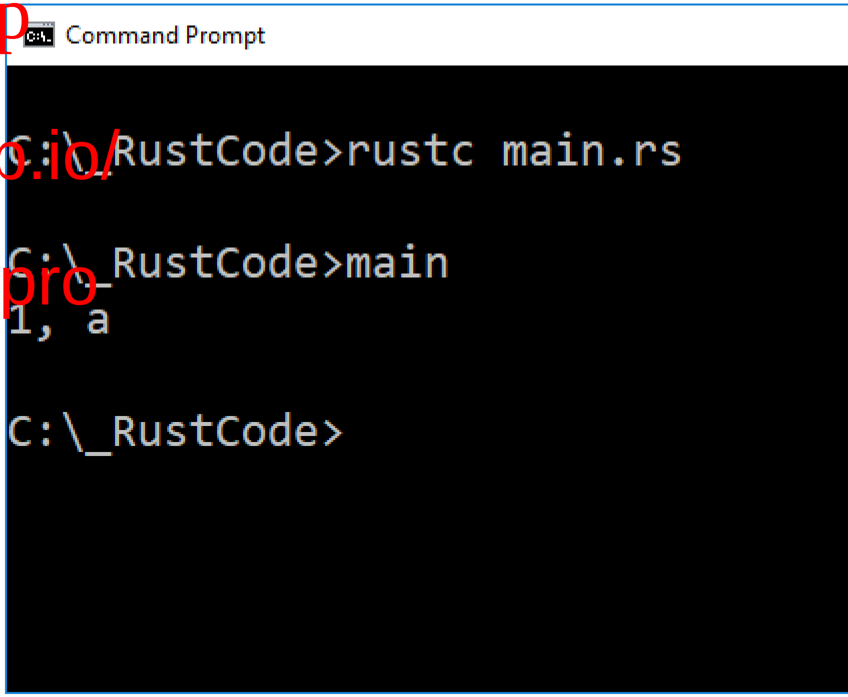
Accessing Elements

Out of bounds:

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

Array of Tuples

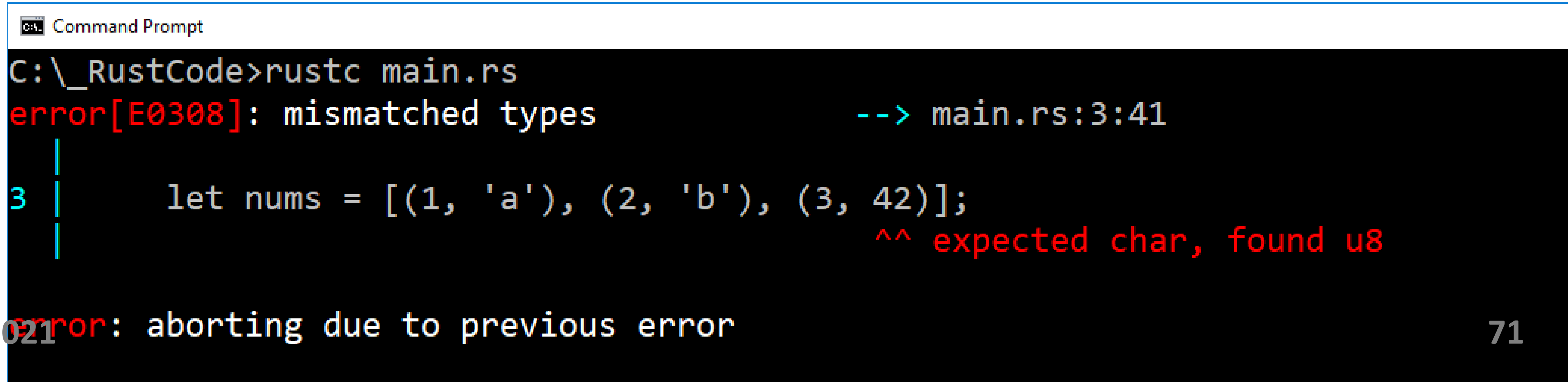
Same rules as Haskell:



```
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



```
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, f

Character – Unicode: 'Z 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

String literals and escape characters are as expected

Functions

We've seen `main()`

- Returns nothing, accepts no arguments.

Convention for naming functions is snake_case.

- Words separated by underscores.

Functions

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

Parameters

`identifier: type`

- Parameters separated by commas.
- Indicating type is ***mandatory***

thing too unusual here

Careful Now…

Statements & Expressions

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

Two types of statements:

- Declaration state
- Expression state)

```
let x = 6;      // This is        ation 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 state
- Expression state)

```
5 + 2;     // This is an ex       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

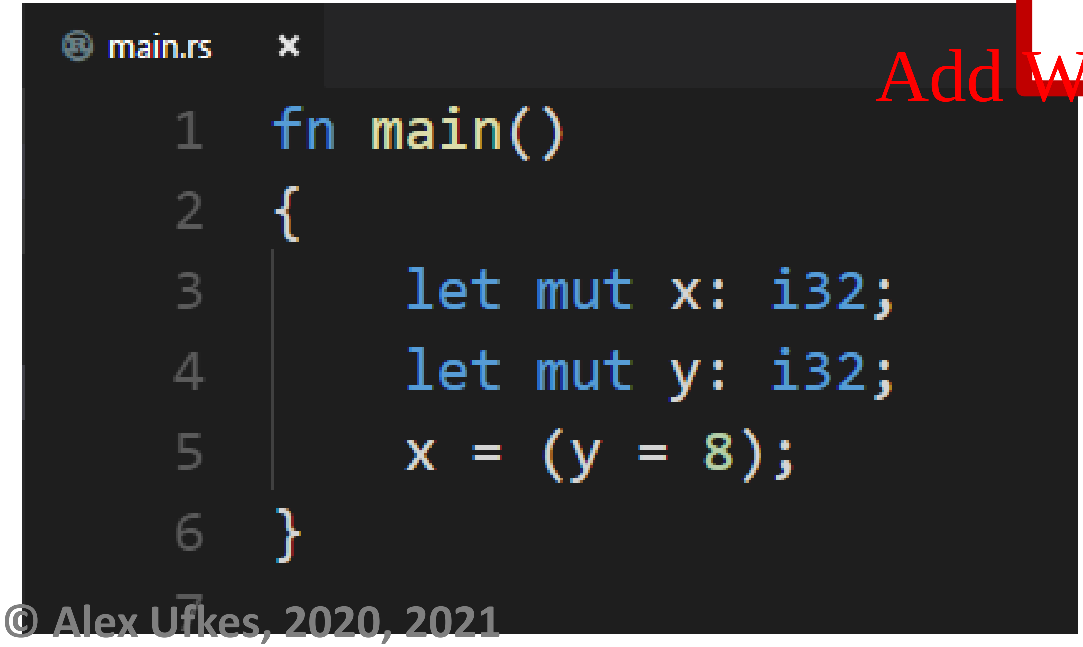
Statements & Expressions

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

Statements & Expressions

OK

Not O at does this error mean?



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

Statements & Expressions

- 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

Here:

ue of x will be 3

of y will be ()

y tuple

Statements & Expressions

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

$\boxed{x = \underbrace{5 + 6}_{\boxed{\text{Expression}}};}$

expression statement

an expression

Expression statement

Decl statement

In fact: $\boxed{\texttt{let x = }\underbrace{\texttt{6}}_{\boxed{\text{Expression}}}\texttt{;}}$

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:

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:

There's a few things going on here:

g to bind a value to y.

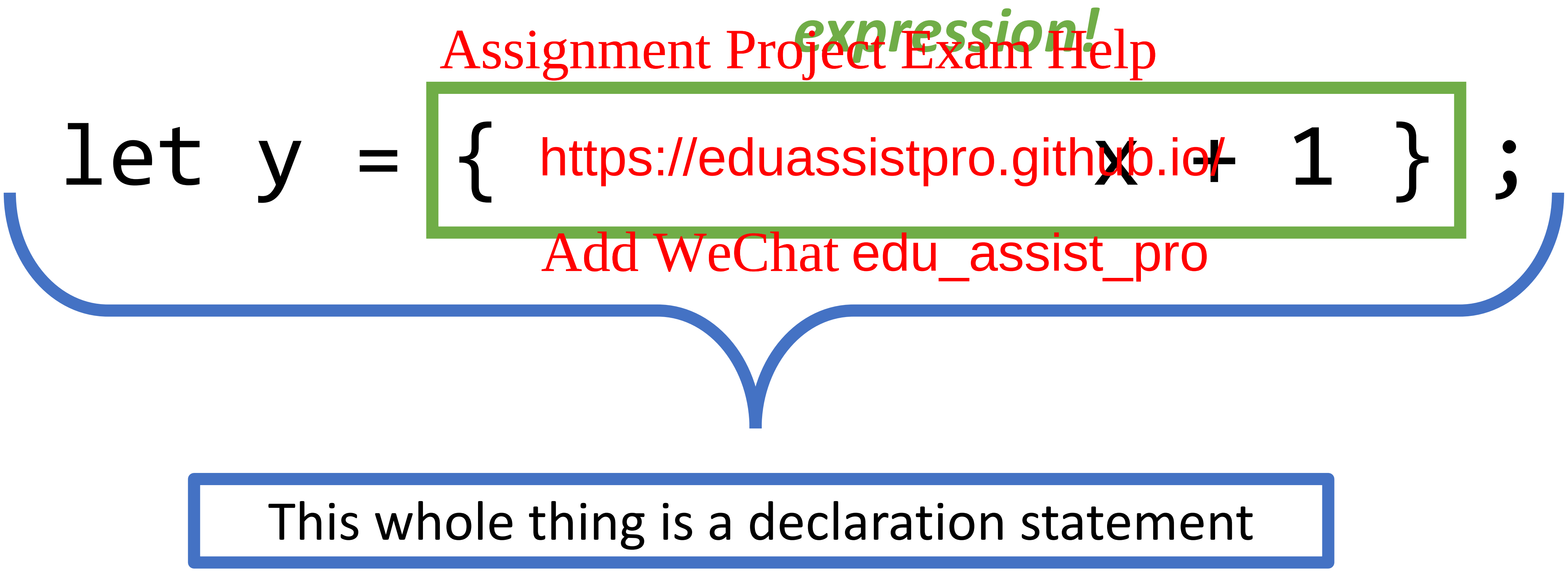
lock { } should evaluate to

s

- 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 = { x + 1 };
```

This whole thing is a declaration statement

Scope Blocks as Expressions

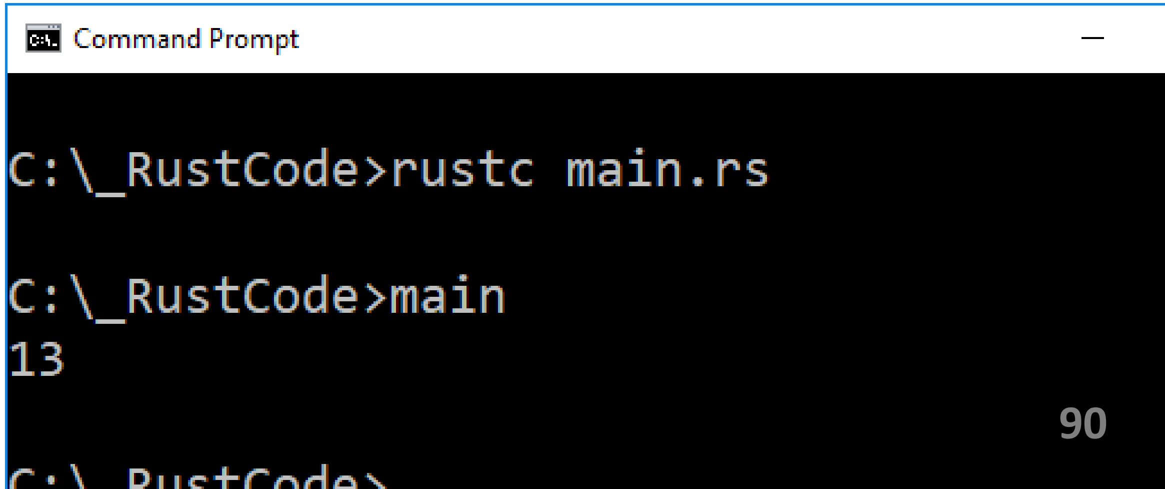
Scope blocks like this are expressions in Rust:

Return Value

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

type

ly indicate return type
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:

Fantastic Rust Reference:

https://doc.rust /second-edition/