C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 10: Rust intro, typing, and control flow

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 C/CPS 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!

© Alex Ufkes, 2020, 2022

Moving on…

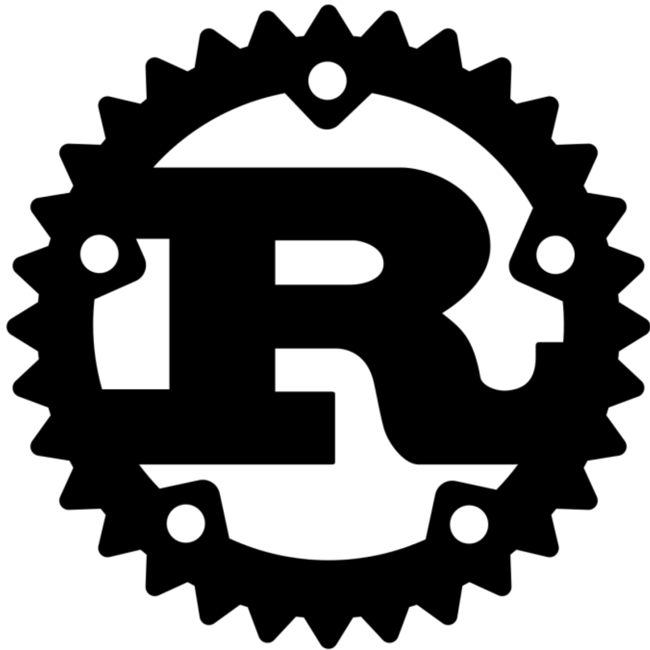
…to imperative.

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

© Alex Ufkes, 2020, 2022

© Alex Ufkes, 2020, 2022

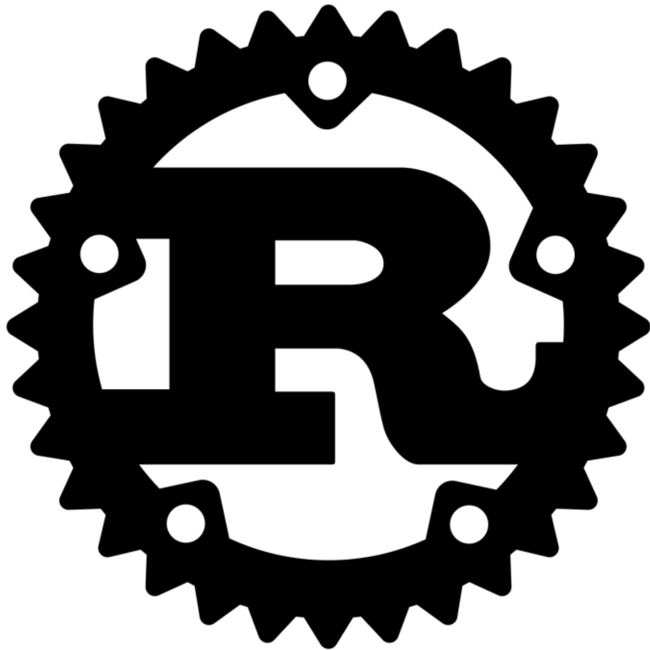
Rust History



- 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

© Alex Ufkes, 2020, 2022

Rust Features

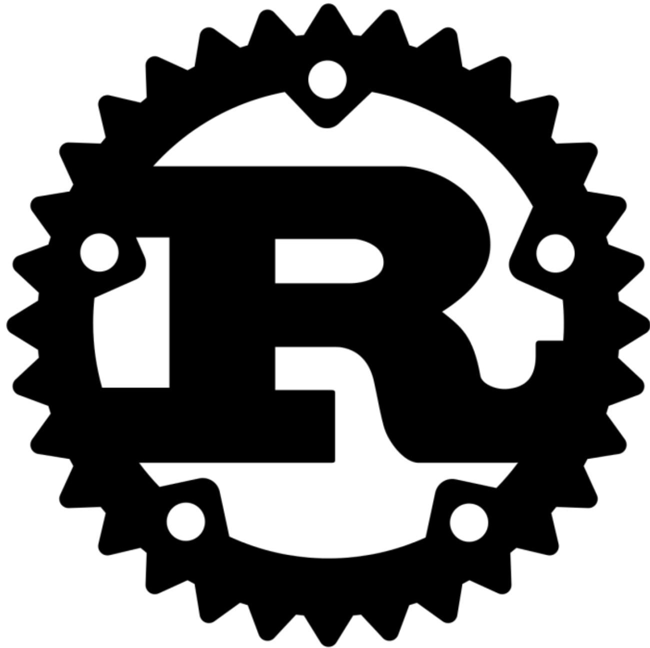


<u>Systems Programming Language</u>:

- 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.

© Alex Ufkes, 2020, 2022

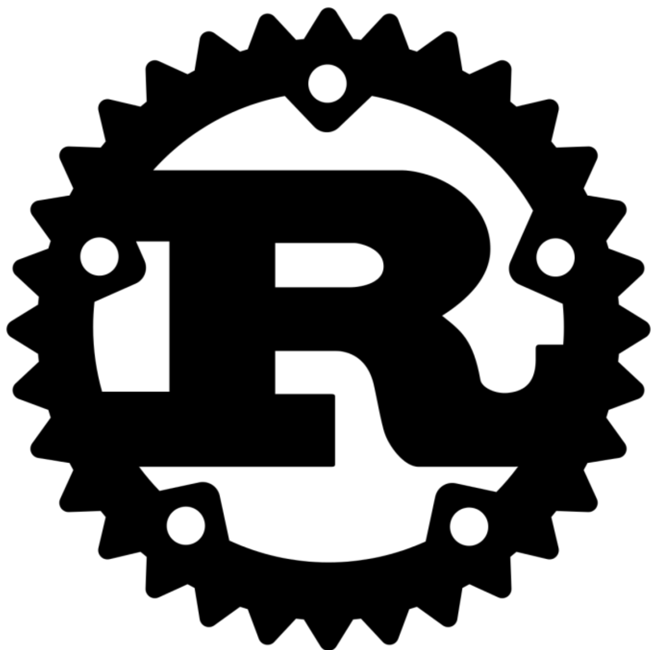
Rust Features



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.

© Alex Ufkes, 2020, 2022

“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 ***try*** 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.

© Alex Ufkes, 2020, 2022

"Null or dangling pointers are not permitted"

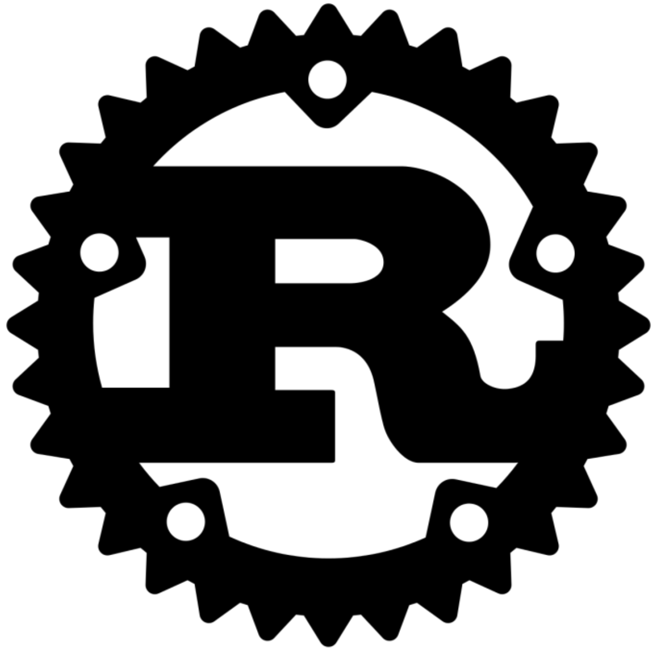
```java
public class Paradigm
{
    public static void main(String[] args)
    {
        int[] nums = {1, 2, 3, 4, 5};

        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 declared without instantiating its object.

© Alex Ufkes, 2020, 2022

Rust Features

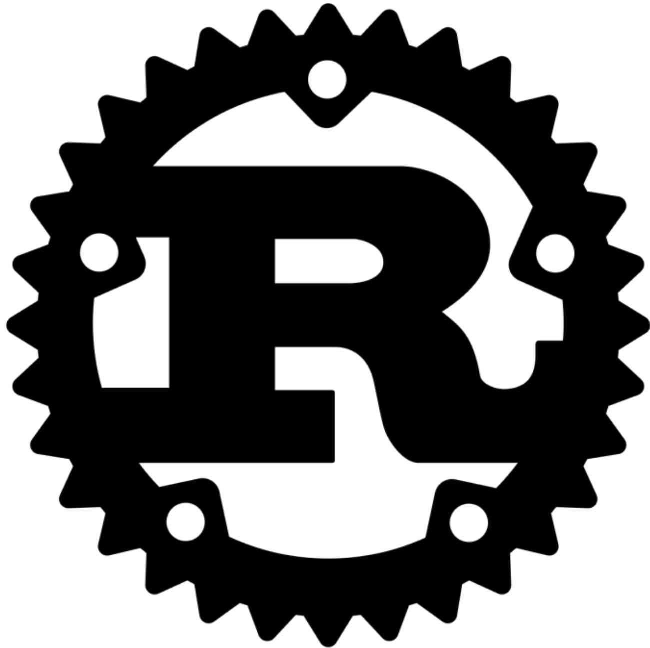


Memory Safety:

- 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?

© Alex Ufkes, 2020, 2022

Rust Features

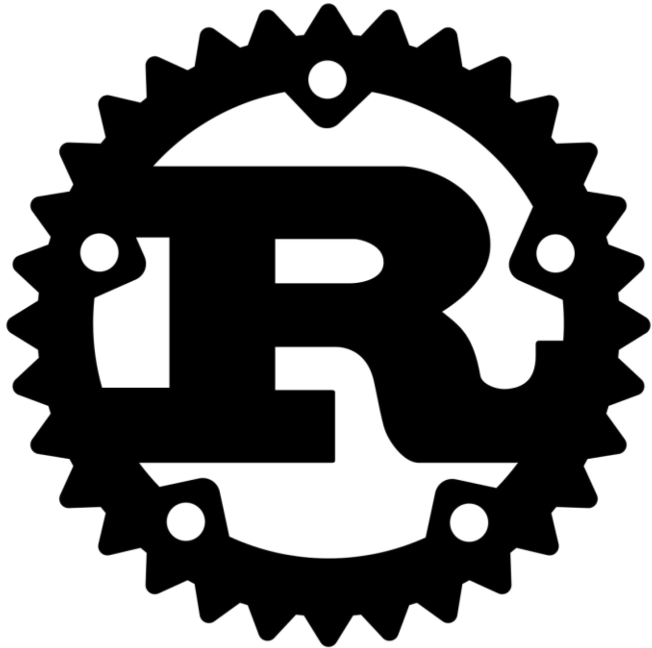


<u>Memory Management</u>:

- 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.

© Alex Ufkes, 2020, 2022

Rust Features

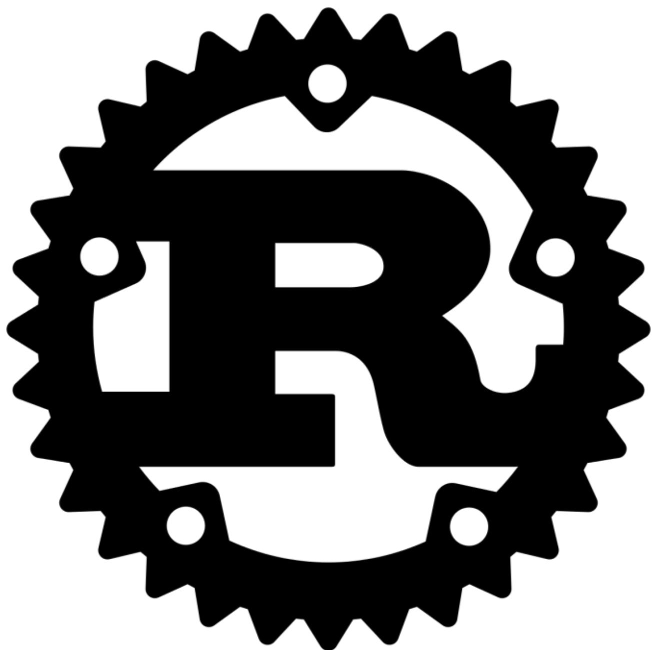


Types and Polymorphism:

- 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.

© Alex Ufkes, 2020, 2022

Rust Features

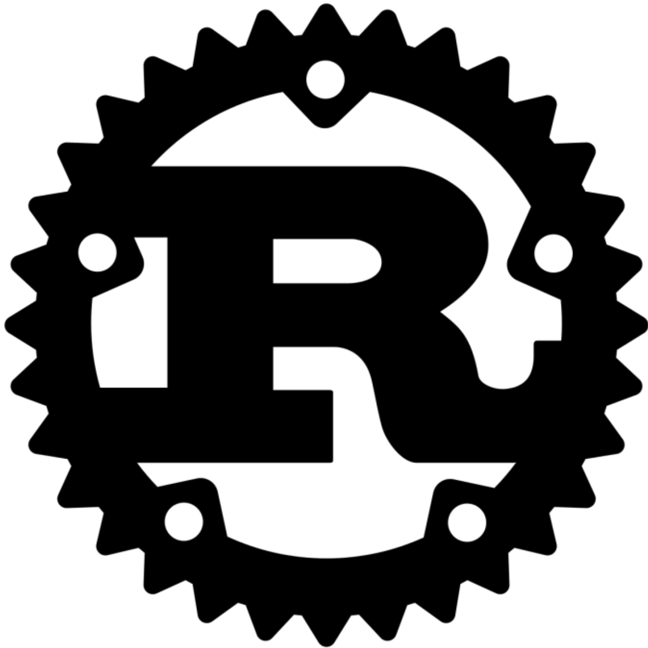


Pattern Matching:

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

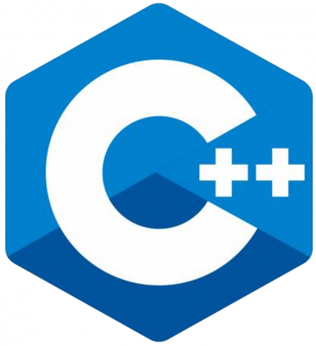
© Alex Ufkes, 2020, 2022

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;
}
```

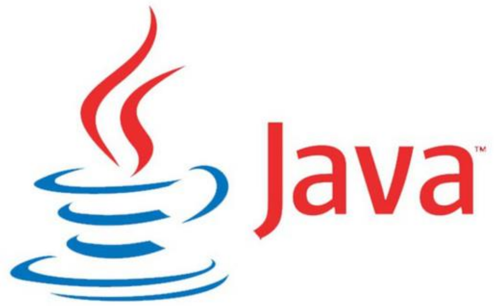
Output

Show output from: Build

```
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
```

© Alex Ufkes, 2020, 2022

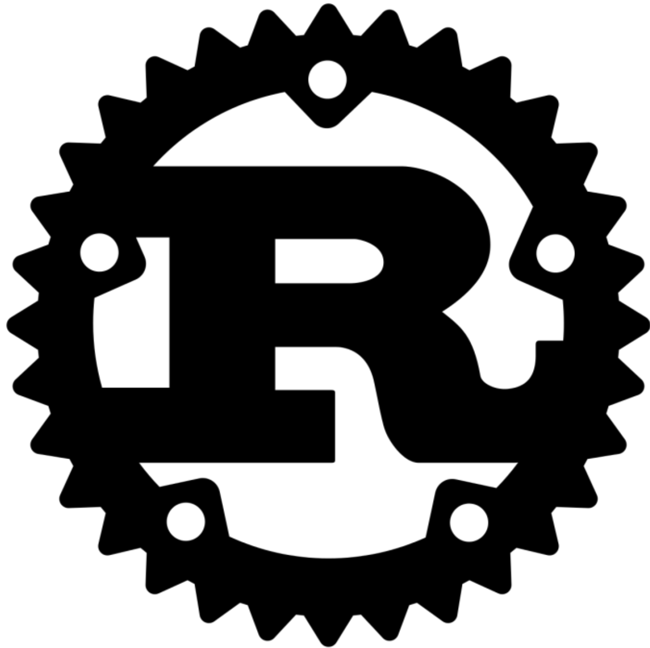



```java
public class Paradigm
{
    public static void main(String[] args)
    {
        float x = 3.1415;
    }
}
```

incompatible types: possible lossy conversion from double to float

© Alex Ufkes, 2020, 2022

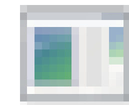
Rust & Safety



No "Undefined Behavior"

- Null pointer dereferencing
  - Attempt to dereference address 0

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

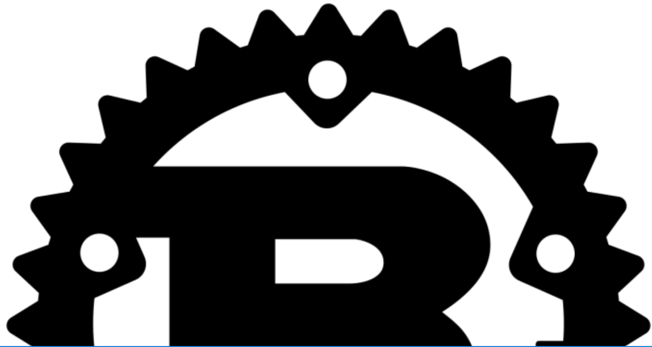


D:\Googl

```
0
Press any
```

© Alex Ufkes, 2020, 2022

Rust & Safety



Quincy 2005 - [Text1 *]

File Edit View Project Debug Tools Window Help

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

Press F1 for Help   Ln

No "Undefined Behavior"

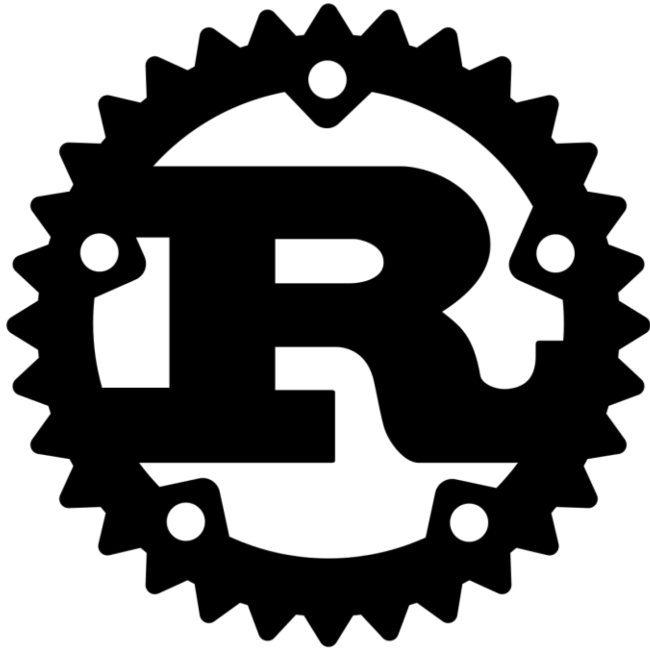
- Null pointer dereferencing
  - Attempt to dereference address 0
- Use of variable before it's initialized
  - In C, we get whatever was in memory before that.
  - Only globals auto-initialize to 0

quincy

```
0
4199232
```

©Alex Ufkes, 2020, 2022

Rust & Safety



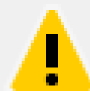
No "Undefined Behavior"

- 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

© Alex Ufkes, 2020, 2022

```c
#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.

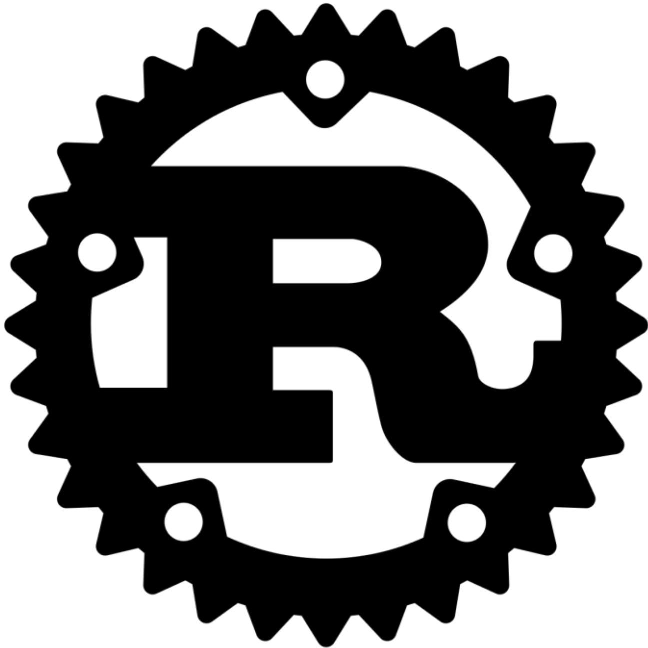
☐ Break when this exception type is thrown

Open Exception Settings

Break | Continue | Ignore

© Alex Ufkes, 2020, 2022

Rust & Safety



No "Undefined Behavior"

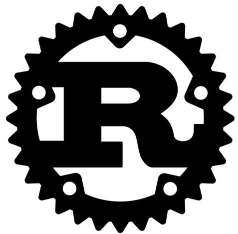
- Signed integer overflow & optimization

$$X+1 > X$$

- 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.

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

Installing Rust

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

Documentation Install Community Contribute

Fork me on GitHub

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

Install Rust 1.26.0

May 10, 2018

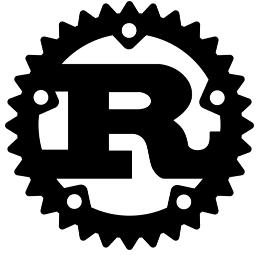
See who's using Rust, and read more about Rust in production.

Featuring

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

Run

© Alex Ufkes, 2020, 2022



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
```

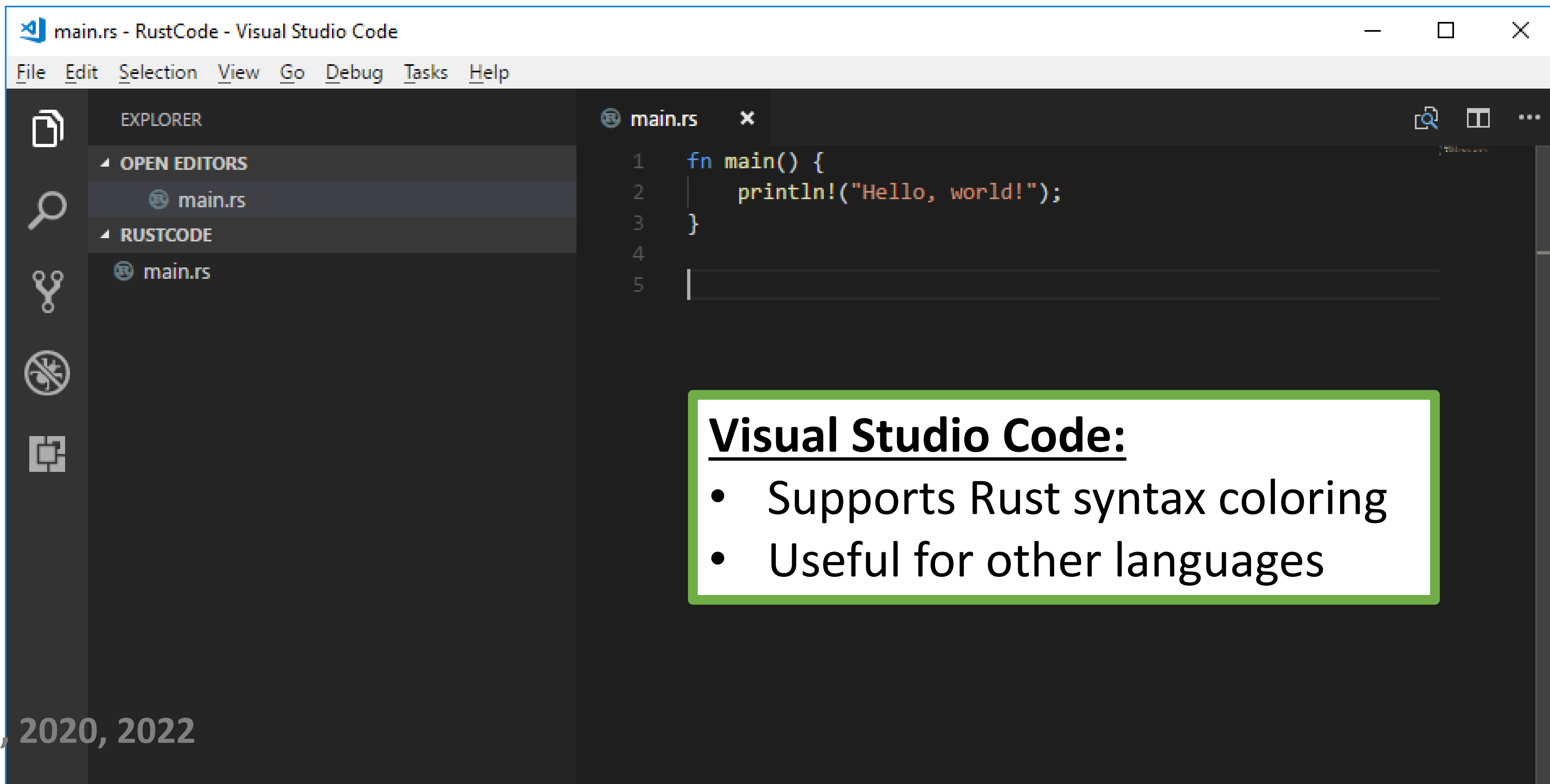
Rust 1.26.0

May 10, 2018

© Alex Ufkes, 2020, 2022

Editing Rust Code

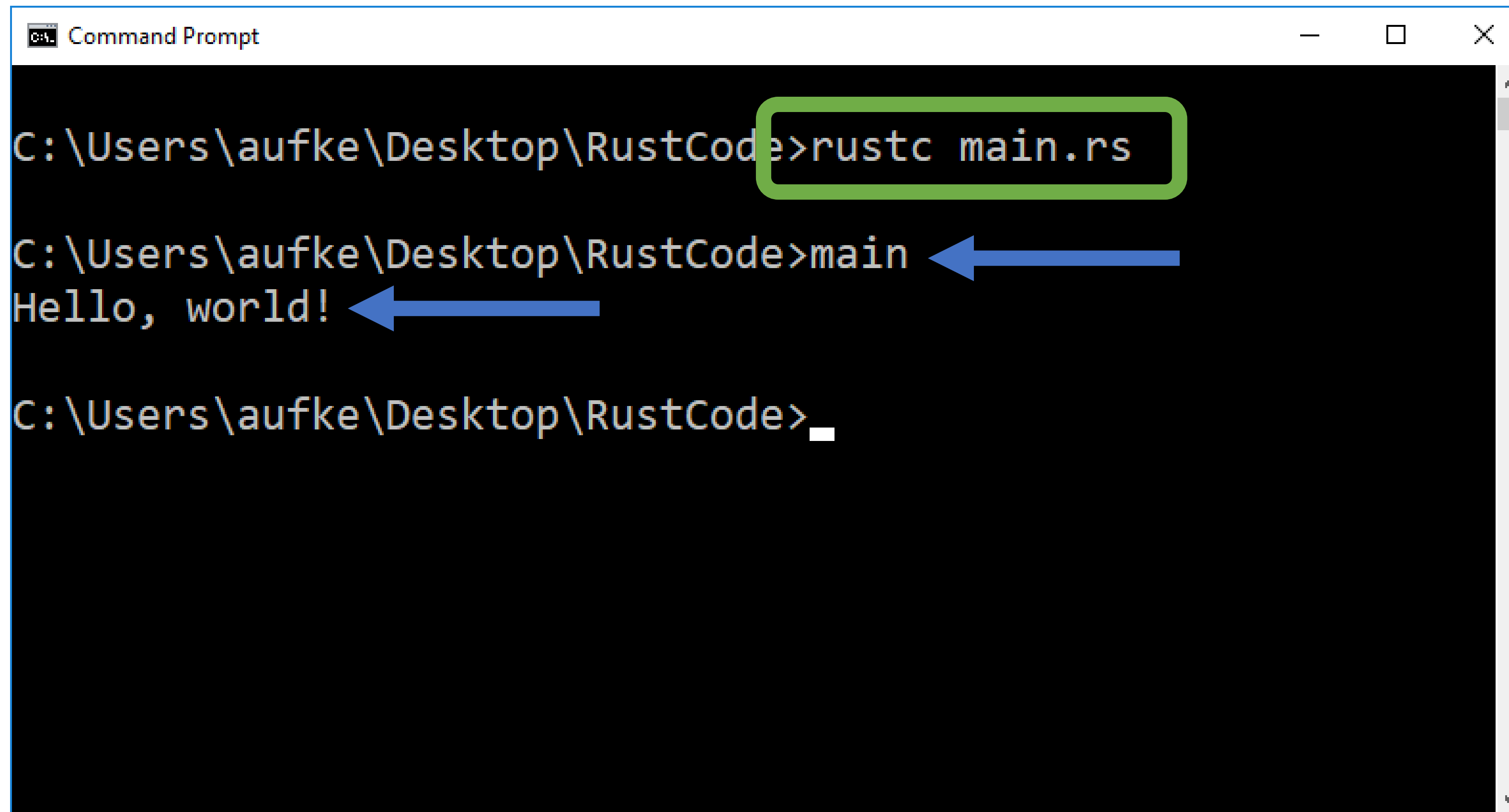
Any text editor will do, but I like VSCode:



© Alex Ufkes, 2020, 2022

Compiling Rust Code

Command Line - **`rustc`**



© Alex Ufkes, 2020, 2022

CCPS506_Rust - Replit

replit.com/@AlexUfkes/CCPS506Rust#main.rs

AlexUfkes / CCPS506_Rust

Invite

Files

main.rs

.replit

main

main.rs

```rust

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



```

Console

Shell

```
> rustc -o main main.rs
main
Hello, world!
>
```

© Al

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.

© Alex Ufkes, 2020, 2022

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

println vs println!

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

© Alex Ufkes, 2020, 2022

Variables

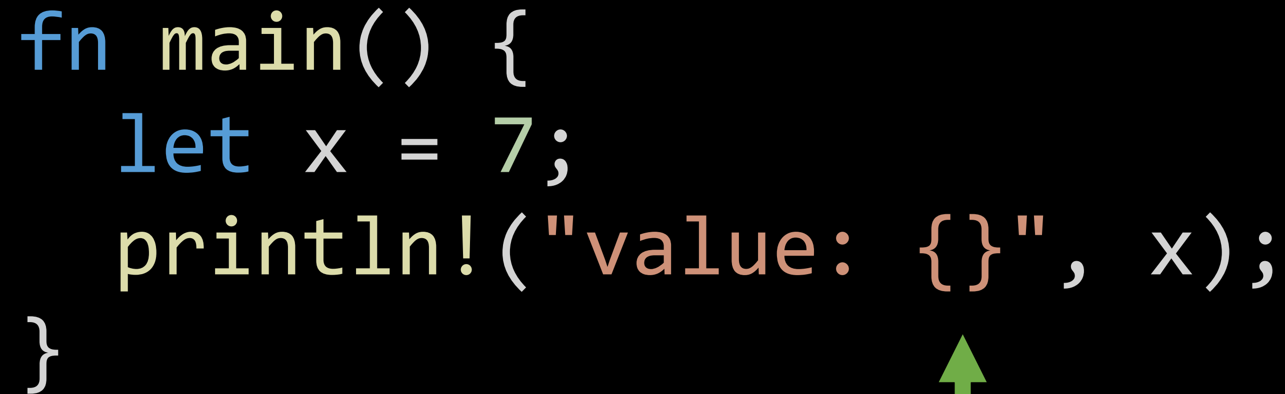
- By default, Rust variables are immutable
- Once initialized, can't change.
- Like `final` or `const` in other languages
- Declare using `let` keyword:

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

Command Prompt

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

C:\_RustCode>main
value: 7

C:\_RustCode>_
```

© Alex Ufkes, 2020, 2022



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

© Alex Ufkes, 2020, 2022

Variables

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

Command Prompt

```
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>_
```

© Alex Ufkes, 2020, 2022

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.

Command Prompt

```
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>
```

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

Constant/Global Variables

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

main.rs

```
1  const BIN: u32 = 2;
2  let x = 2;
3  fn main() {
4      println!("Base: {}", BIN);
5      println!("Base: {}", x);
6  }
7
```

Command Prompt

```
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>
```

© Alex Ufkes, 2020, 2022

Shadowing

Variables with the same name?

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

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

BAD

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

OK

© Alex Ufkes, 2020, 2022

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

© Alex Ufkes, 2020, 2022

Shadowing

Variables with the same name?

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

```
Command Prompt

C:\_RustCode>rustc main.rs

C:\_RustCode>main
x: 8

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Shadowing

Variables with the same name?

main.rs

```
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.

© Alex Ufkes, 2020, 2022

Shadowing VS mut

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

main.rs

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

Command Prompt

```
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.

© Alex Ufkes, 2020, 2022

Shadowing VS mut

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

main.rs

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

Command Prompt

```
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.

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

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

© Alex Ufkes, 2020, 2022

Specify Type?

Rust has type inference, but we can be explicit:

main.rs

```
1 fn main() {
2     let x: u8 = 3;
3     let y: i64 = 5;
4     let z: isize = 999;
5     println!("x: {}", x);
6     println!("y: {}", y);
7     println!("z: {}", z);
8 }
9
```

Command Prompt

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

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

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

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.

main.rs

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

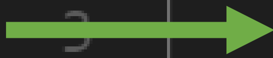
© Alex Ufkes, 2020, 2022

Scalar Types: Floating Point

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

main.rs

```
1  fn main() {
2      let x: f32 = 1.0/3.0;
3      let y: f64 = 1.0/3.0;
4      println!("x: {}", x);
5      println!("y: {}", y);
6  }
7
```



Command Prompt

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

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

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Numeric Operations

```
main.rs

1  fn main() {
2      let r1 = 2 + 3
3      let r2 = 3/4;
4      let r3 = 2 % 3
5      println!("r1:
6      println!("r2:
7      println!("r3:
8  }
9
```

```
Command Prompt
x: 0.33333334
y: 10000

C:\_RustCode>rustc main.rs
error[E0277]: cannot mod `{integer}` by `{float}`
 --> main.rs:4:16
  |
4 |     let r3 = 2 % 3.0;
  |                ^ no implementation for `{intege
r} % {float}`
  |
  = help: the trait `std::ops::Rem<{float}>` is not
 implemented for `{integer}`
```

© Alex Ufkes, 2020, 2022

Numeric Operations

main.rs

```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);
}
```

Command Prompt

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

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

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Mixed Expressions?

main.rs

```
1  fn main() {
2      let r1 = 3/4;
3      let r2 = 3/4.0;
4      let r3: f64 = 3/4;
5      println!("r1: {}", r
6      println!("r2: {}", r
7      println!("r3: {}", r
8  }
9
```

Command Prompt

```
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.

© Alex Ufkes, 2020, 2022

Mixed Expressions?

main.rs

```
1  fn main() {
2      let r1: f64 = 3/4
3      let r2 = 3 as f64/
4      println!("r1: {}"
5      println!("r2: {}"
6  }
7
```

Command Prompt

```
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
```

© Alex Ufkes, 2020, 2022

Mixed Expressions?

main.rs

```
1  fn main() {
2      //let r1: f64 = 3/4;
3      let r2 = 3 as f64/4 as f64;
4      //println!("r1: {}", r1);
5      println!("r2: {}", r2);
6  }
7
```

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

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

Command Prompt

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

C:\_RustCode>main
r2: 0.75

C:\_RustCode>
```

Finally!

© Alex Ufkes, 2020, 2022

Mixed Expressions?

Division may truncate, good reason to avoid implicit conversion…

Command Prompt

```
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>
```

main.rs

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

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

```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);
    }
}
```

BlueJ: Terminal Window - HelloWorld

Options

```
2111111111
2111111168
```

Can only ent

IEEE-754

HISTORY.COM

© Alex Ufkes, 2020, 2022

Scalar Types: Boolean

`true`, `false`. Easy:

main.rs

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

Command Prompt

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

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

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Scalar Types: Characters

Rust supports Unicode:

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

Command Prompt

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

C:\_RustCode>main
c1: Z
c2: Å

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

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.

© Alex Ufkes, 2020, 2022

Accessing Elements

De-structuring!

```
1  fn main()
2  {
3      let tup = (42, 3.141592, '!');
4      let (x, y, z) = tup;
5
6      println!("{}, {}, {}", x, y, z);
7  }
8
```

```
Command Prompt

C:\_RustCode>rustc main.rs

C:\_RustCode>main
42, 3.141592, !

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Accessing Elements

Can also access directly:

```
main.rs
1 fn main()
2 {
3     let tup = (42, 3.141592, '!');
4     println!("{}, {}, {}", tup.0, tup.1, tup.2);
5 }
6
```

```
Command Prompt

C:\_RustCode>rustc main.rs

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

Can we go out of bounds?

© Alex Ufkes, 2020, 2022

Accessing Elements

Out of bounds:

```
main.rs

1  fn main()
2  {
3      let tup = (42, 3.141592, '!');
4      println!("{}", tup.0);
5      println!("{}", tup.1);
6      println!("{}", tup.2);
7      println!("{}", tup.3);
8  }
9
```

```
Command Prompt
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);
}
```

```
Command Prompt
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.

© Alex Ufkes, 2020, 2022

Compound Types

Arrays

main.rs

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

Command Prompt

```
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.

© Alex Ufkes, 2020, 2022

Accessing Elements

Out of bounds:

main.rs

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

Command Prompt

```
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:

main.rs

```
1  fn main()
2  {
3      let nums = [(1, 'a'), (2, 'b'), (3,
4
5      println!("{}, {}", nums[0].0, nums[
6  }
7
```

Command Prompt

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

C:\_RustCode>main
1, a

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Array of Tuples

Same rules as Haskell: Tuple types must be the same

main.rs

```
1  fn main()
2  {
3      let nums = [(1, 'a'), (2, 'b'), (3, 42)];
4
5
6  }
7
```

Command Prompt

```
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
```

© Alex Ufkes, 2020, 2022

Types & Literals: Summary

<u>4 Scalar types:</u>

Integer – `u8, u16, u32, u64, usize, i8, i16,` **<u>`i32`</u>**`, i64, isize`
Floating Point – `f32,` **<u>`f64`</u>**
Boolean – `bool` (true, false)
Character – Unicode: `'Z', 'a', '&', '\u{00C5}', etc`

<u>2 Compound types:</u>

Tuple – heterogeneous
Arrays – homogeneous

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

© Alex Ufkes, 2020, 2022

Strings

main.rs

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

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

Command Prompt

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

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

C:\_RustCode>
```

String literals and escape characters are as expected

© Alex Ufkes, 2020, 2022

FUNCTIONs

Functions

We've seen `main()`

```
main.rs
1  fn main()
2  {
3      like_main_but_not_as_good ();
4  }
5
6  fn like_main_but_not_as_good ()
7  {
8      println!("Hello World!");
9  }
10
```

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

© Alex Ufkes, 2020, 2022

Functions

main.rs

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

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

Command Prompt

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

C:\_RustCode>main
Hello World!

C:\_RustCode>_
```

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

© Alex Ufkes, 2020, 2022

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

Command Prompt

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

C:\_RustCode>main
5
5, 3.14

C:\_RustCode>_
```

© Alex Ufkes, 2020, 2022

Careful Now…

```
main.rs

1  fn main()
2  {
3      print_val (5);
4      print_two_vals (5, 3);
5  }
6
7  fn print_val (n: i32)
8  {
9      println!("{}", n);
10 }
11
12 fn print_two_vals (n1: i32, n2: f64)
13 {
14     println!("{}, {}", n1, n2);
15 }
16
```

```
Command Prompt

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`.
```

© Alex Ufkes, 2020, 2022

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);
```

© Alex Ufkes, 2020, 2022

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

© Alex Ufkes, 2020, 2022

Statements & Expressions

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

main.rs

```
1  fn main()
2  {
3      let x = (let y = 6);
4  }
5
```

Command Prompt

```
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
```

© Alex Ufkes, 2020, 2022

Statements & Expressions

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

OK

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

Not OK… but what does this error mean?

```
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 `()`
```

© Alex Ufkes, 2020, 2022

Statements & Expressions

main.rs

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

Command Prompt

```
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 `()`
```

- 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`**!

© Alex Ufkes, 2020, 2022

Statements & Expressions

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

}
```

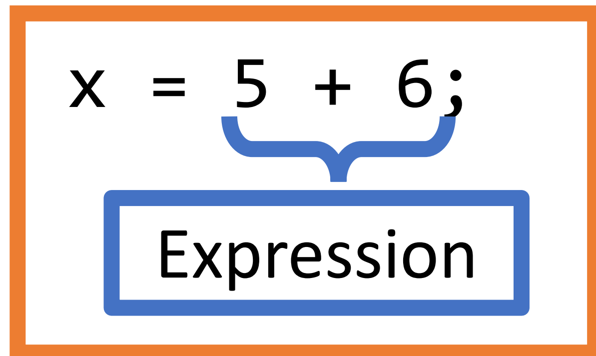
Here:

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

© Alex Ufkes, 2020, 2022

Statements & Expressions

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

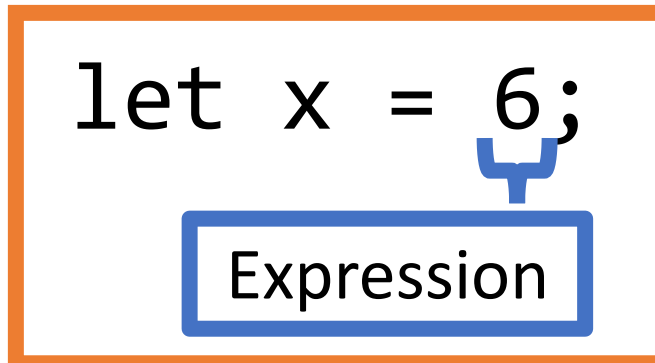


Expression statement

```
                   // This is an expression statement
                   // containing an expression
```

Declaration statement

In fact:



© Alex Ufkes, 2020, 2022

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

© Alex Ufkes, 2020, 2022

Scope Blocks as Expressions

Scope blocks like this are expressions in Rust:

```
main.rs  ×   Lab4.rs
1  fn main()
2  {
3      let x = 5;
4
5      let y = {
6          let z = 3;
7          z + 1
8      };
9
10     println!("{} {}", x, y);
11 }
12
```

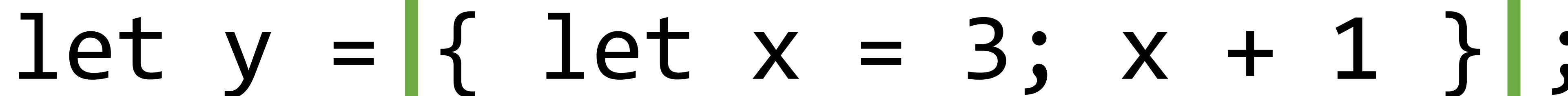
There's a few things going on here:

- 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.

© Alex Ufkes, 2020, 2022

Scope Blocks as Expressions

Scope blocks like this are expressions in Rust:



This whole thing is a declaration statement

© Alex Ufkes, 2020, 2022

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>
```

© Alex Ufkes, 2020, 2022

Return Value

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

main.rs

```
1  fn main()
2  {
3      println!("{}", plus_five(8));
4  }
5
6  fn plus_five (n: i32) -> i32
7  {
8      n + 5
9  }
10
```

-> type

- Explicitly indicate return type
- Result of expression gets returned

Command Prompt

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

C:\_RustCode>main
13

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Return Value

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

```
main.rs
1  fn main()
2  {
3      println!("{}",
4  }
5
6  fn plus_five (n: i
7  {
8      n + 5;
9  }
10
```

```
Command Prompt
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 `()`
```

© Alex Ufkes, 2020, 2022

Control Flow

© Alex Ufkes, 2020, 2022

if / else

main.rs

```rust
fn main()
{
    let num = 3;

    if num > 5 {
        println!("Greater than 5!");
    }
    else {
        println!("Not that thing I just said");
    }

}
```

- As with other imperative languages, the else is optional.
- Recall that this is not the case with Haskell!
- We were required to have a complete if-then-else

Command Prompt

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

C:\_RustCode>main
Not that thing I just said

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Boolean Conditions?

Mandatory.

In C/C++ (and Elixir, with caveats):

- Non-zero values are "truthy".
- Only 0/nil considered false.

```
if (3.141592)
  cout << "Valid!" << endl;
```

In Java (and Haskell, Rust):

- Conditions must be Boolean

```
if (3.141592)
  System.out.println(
    "Compile Error");
```

Converting non-Boolean to Boolean requires implicit conversion, which, as we've seen, Rust does not do.

© Alex Ufkes, 2020, 2022

if / else if / else

main.rs

```rust
fn main()
{
    let temp = 33;

    if temp < 0 {
        println!("Frozen");
    }
    else if temp < 100 {
        println!("Liquid");
    }
    else {
        println!("Boiling");
    }
}
```

- As we'd expect.
- We use { } even though there's only one statement per branch
- This is required.
- Why? Rust treats these as blocks whose last line can be an expression.

© Alex Ufkes, 2020, 2022

if / else if / else

main.rs

```rust
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

if / else if / else

```rust
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

- **`let state = {…};`** is a statement
- **`{…}`** is an expression that will evaluate to a string. **`if`** == expression!
- "Frozen", "Liquid", or "Boiling"
- Each option is in a scope block { }
- The value of a scope block is the last expression
- Leaving the **`;`** off makes these strings expressions.

© Alex Ufkes, 2020, 2022

if / else if / else

```
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

Command Prompt

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

C:\_RustCode>main
Water is Liquid!

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Problem?

```
fn main()
{
    let temp = 33;

    let num = if temp > 50 { 33.33 }
              else { 99 };

    println!("num: {}", num);
}
```

```
C:\_RustCode>rustc main.rs
error[E0308]: if and else have incompatible types
 --> main.rs:5:15
  |
5 |         let num = if temp > 50 { 33.33 }
  |                                  ^
  |  _______________
6 | |                  else { 99 };
  | |_________________________^ expected floating-point variable, found integral variable
  |
  = note: expected type `{float}`
             found type `{integer}`
```

Might return float, might return int

Remember: Strong, static typing. No implicit conversion!

© Alex Ufkes, 2020, 2022

© Alex Ufkes, 2020, 2022

Looping

```rust
fn main()
{
    loop {
        println!("Again!");
    }
}
```

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

C:\_RustCode>main_
```

```
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
^C
C:\_RustCode>_
```

Just like `while(true){}` in Java

© Alex Ufkes, 2020, 2022

Conditional Looping: `while`

```rust
fn main()
{
    let mut n = 1;

    while n <= 10
    {
        println!("{}", n);
        n += 1;
    }
}
```

- Similar in form to other imperative languages.
- Rust understands +=

```
Command Prompt

C:\_RustCode>main
1
2
3
4
5
6
7
8
9
10

C:\_RustCode>
```

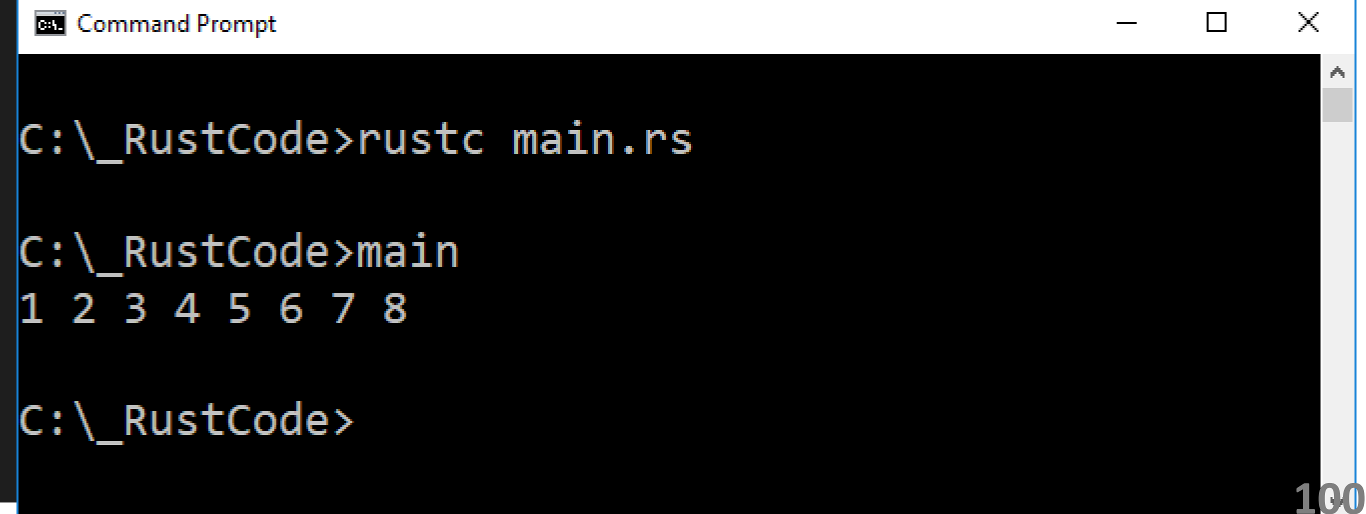
© Alex Ufkes, 2020, 2022

Conditional Looping: for

Just like a for loop in Python:

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8];

    for elem in nums.iter()
    {
        print!("{} ", elem);
    }
    println!();
}
```

- Invoke iter() method of array nums
- **elem** takes the value of each element in the array.
- Safe! Never go out of bounds.



© Alex Ufkes, 2020, 2022

Conditional Looping: for

Use .. to create a range

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    for i in (0..10).rev()
    {
        print!("{} ", nums[i]);
    }
    print!("\nLIFTOFF!\n");
}
```

- Create a *Range* containing 0 to ***9***
- Top of range not included!
- Just like range() in Python

© Alex Ufkes, 2020, 2022

A loop is a loop is a loop

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
warning: comparison is useless due to type limits
 --> main.rs:8:12
  |
8 |         if i < 0 { break; }
  |            ^^^^^
  |
  = note: #[warn(unused_comparisons)] on by defaul
t


C:\_RustCode>
```

Wait, *what?*

Wait, what?

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7,
    let mut i = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

- We didn't specify the type of **`i`**, but shouldn't it default to `i32`?
- Rust infers type, `i32` should be default.
- **HOWEVER!**
- Rust doesn't allow signed integers to be used as array indexes!
- It inferred the type as unsigned! Thus checking less than zero is pointless.

```
C:\_RustCode>rustc main.rs
warning: comparison is useless due to type limits
 --> main.rs:8:12
  |
8 |         if i < 0 { break; }
  |            ^^^^^
  |
```

© Alex Ufkes, 2020, 2022

Rust doesn't allow signed integers to be used as array indexes!

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i: i32 = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0277]: the trait bound `i32: std::slice::SliceIndex<[{integer}]>` is not satisfied
 --> main.rs:9:23
  |
9 |         print!("{} ", nums[i]);
  |                       ^^^^^^^ slice indices are of type `usize` or ranges of `usize`
  |
  = help: the trait `std::slice::SliceIndex<[{integer}]>` is not implemented for `i32`
  = note: required because of the requirements
```

© Alex Ufkes, 2020, 2022

Need to adjust our logic a bit…

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i = 9;

    loop
    {
        print!("{} ", nums[i]);
        if i == 0 { break; }
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

Command Prompt

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

C:\_RustCode>main
10 9 8 7 6 5 4 3 2 1
LIFTOFF!

C:\_RustCode>
```

© Alex Ufkes, 2020, 2022

Fantastic Rust Reference:

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

© Alex Ufkes, 2020, 2022

© Alex Ufkes, 2020, 2022