Rust programming

Course – 1

Gavrilut Dragos

Agenda for today

1. Administrative
2. Intro
3. First rust program
4. Basic Types
5. Variables
6. Operators
7. Functions & Expression Statements
8. Basic statements (if, while, loop, ….)

Administrative

Administrative

Overview:

- Course web page: https://gdt050579.github.io/rust_course_fii/
- Grading: Gauss-like system (check out our Administrative page for more details)

Examination type:

- A lab project →50 points (from week 8)
- Course examination →30 points
- Lab activity →7 points (week 1 to week 7)

Minimal requirements:

- Lab (activity + project) →20 points
- Course examination →10 points

Intro

What is Rust

Rust is an open-source general programming language that focuses on performance and safety (memory safety / type safety). It is primarily used for building command line tools, web applications, server apps or to be used in embedded systems.

Resource:

- Linux & Mac/OSX: run `curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh`
- GitHub repo: https://github.com/rust-lang/rust
- Windows install link: https://www.rust-lang.org/tools/install
- Documentation: https://doc.rust-lang.org/book/
- Quick install: https://rustup.rs/
- Official site: https://www.rust-lang.org/

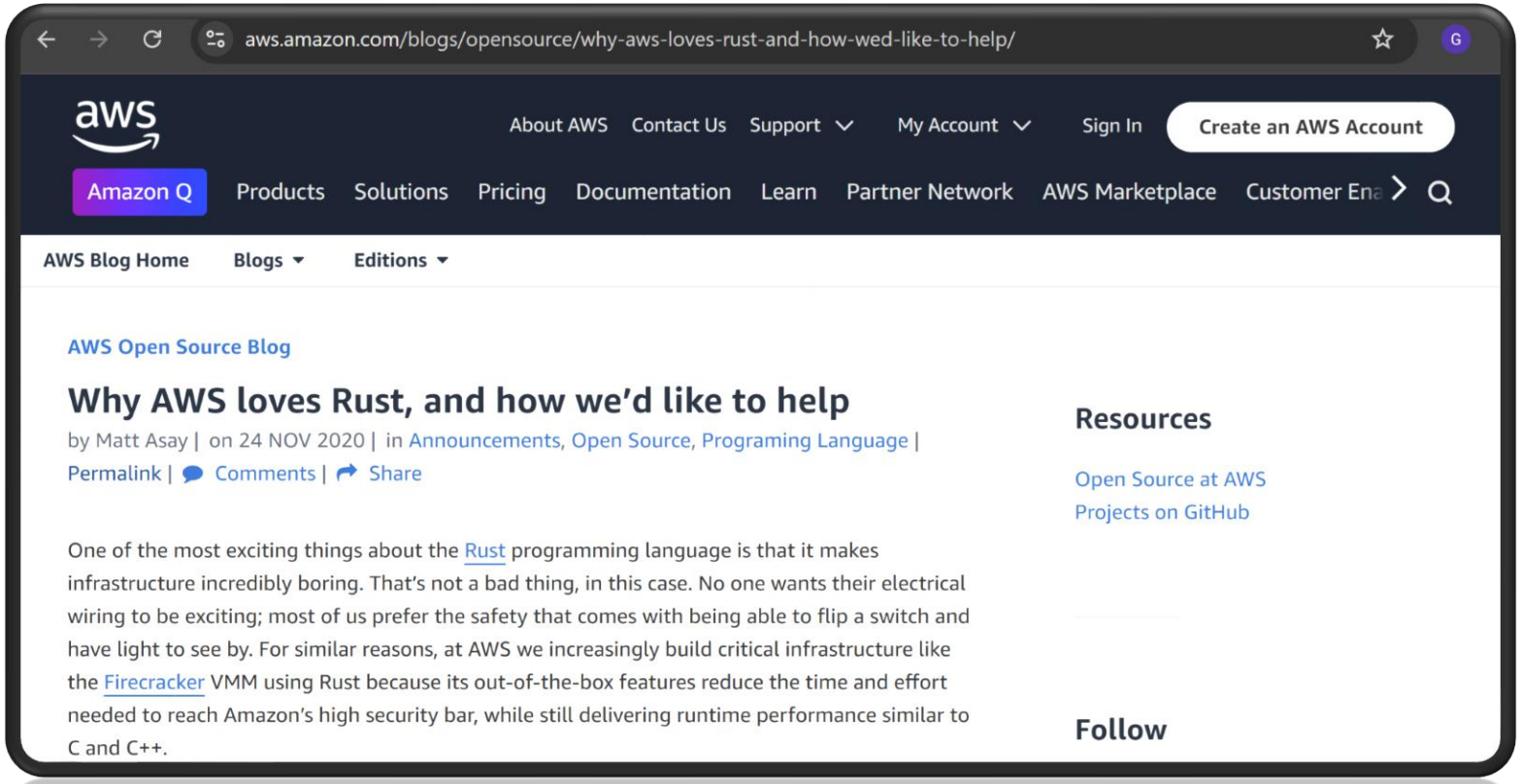
Rust latest version: 1.90.0 *(18.Sep.2024)*

Rust History

- 2006 → started as a project develop in Mozilla by Graydon Hoare
- 2010 → officially announced as a project
- 2015 → Rust 1.0 (first stable released announce)
- 2021 → Rust Foundation is formed, and the project is no longer maintained solely by Mozilla. Companies that are part of Rust Foundations are: AWS, Google, Huawei, Microsoft and Mozilla
- 2022 → **Linus Torvalds** announce that Rust is probably going to be used in Linux Kernel in the near future

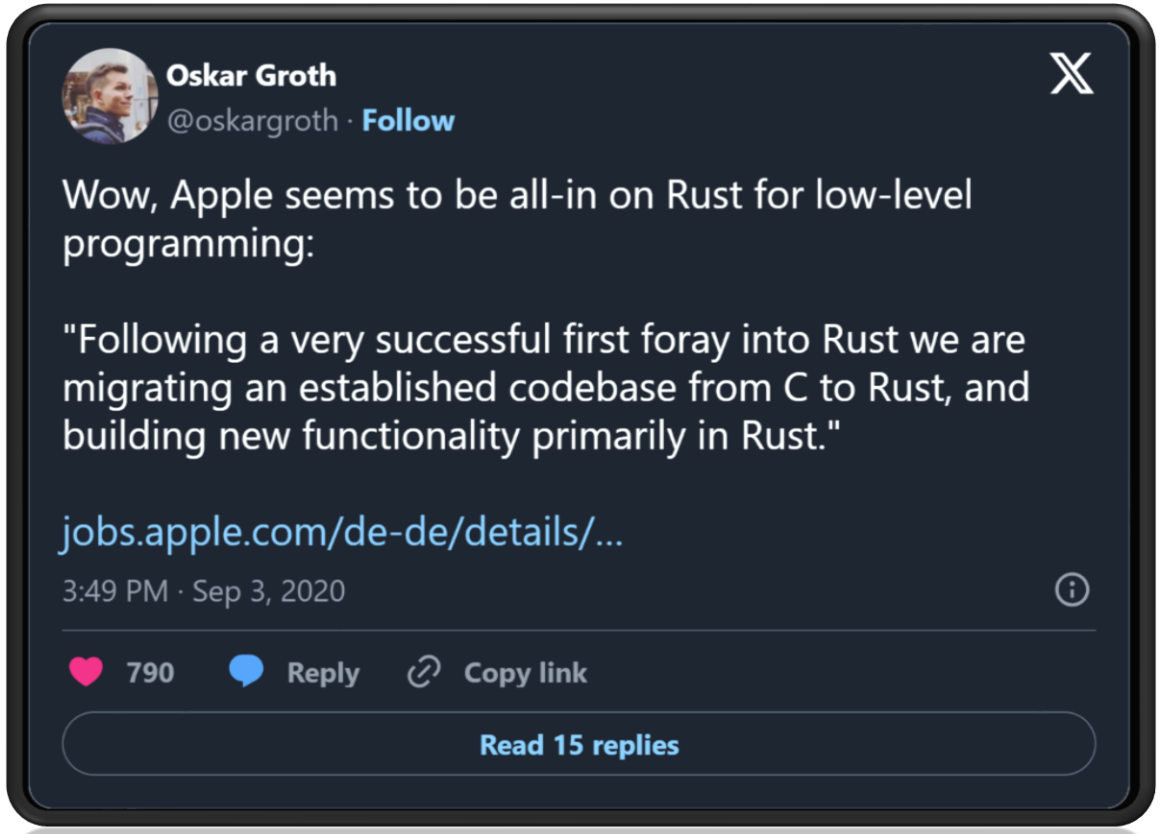
Rust History

- 2020 → Amazon announced its implication in using Rust as a language for various project (AWS FireCracker being one of them)



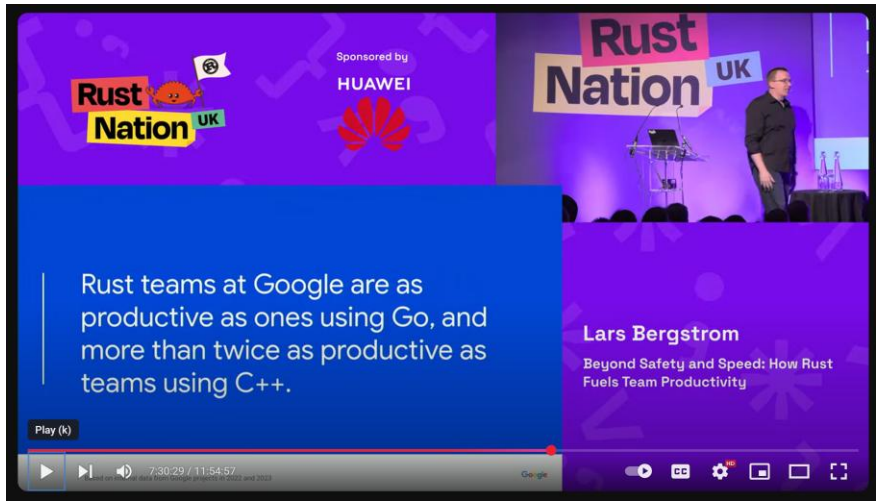
Rust History

- 2020/Sep → While not confirmed by Apple, there are roomers that Apple is also using Rust internally



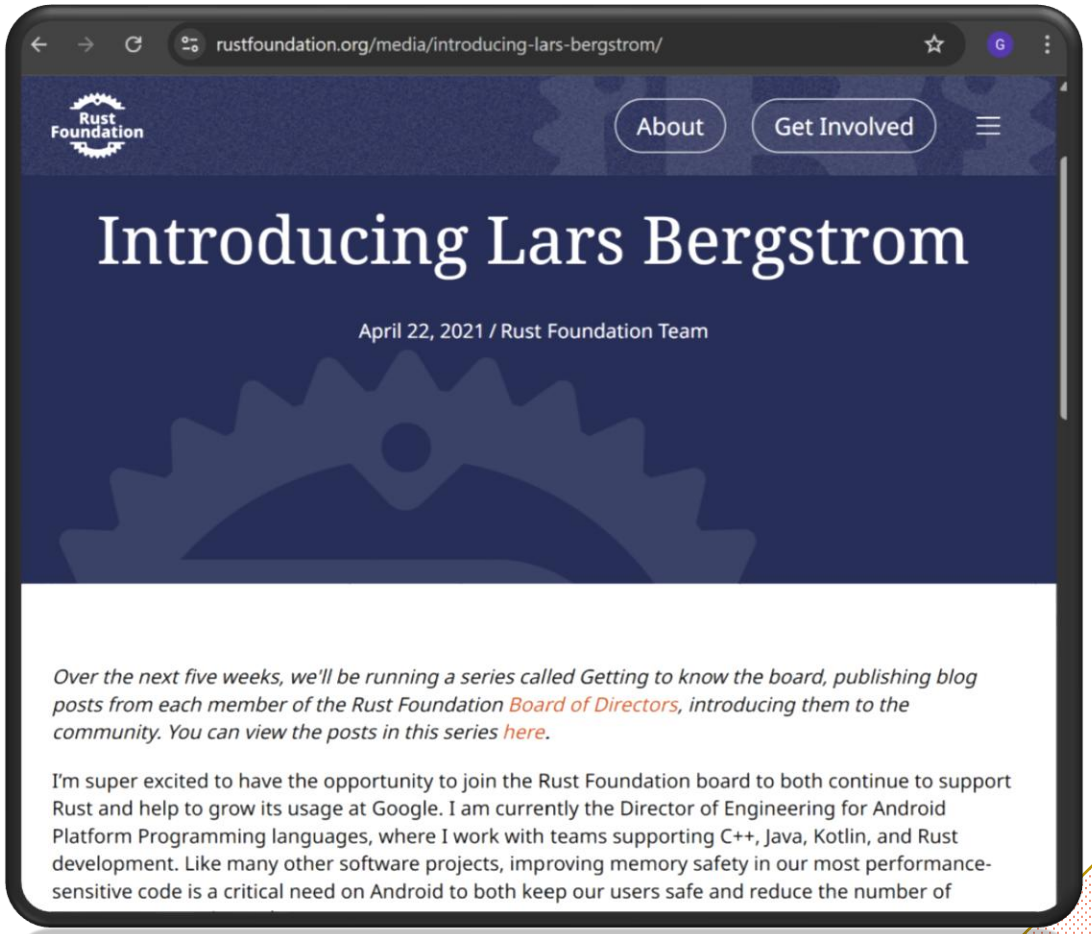
Rust History

- 2021 → Google joins Rust Foundation with the director of Engineering for the Android Platform – Lars Bergstrom



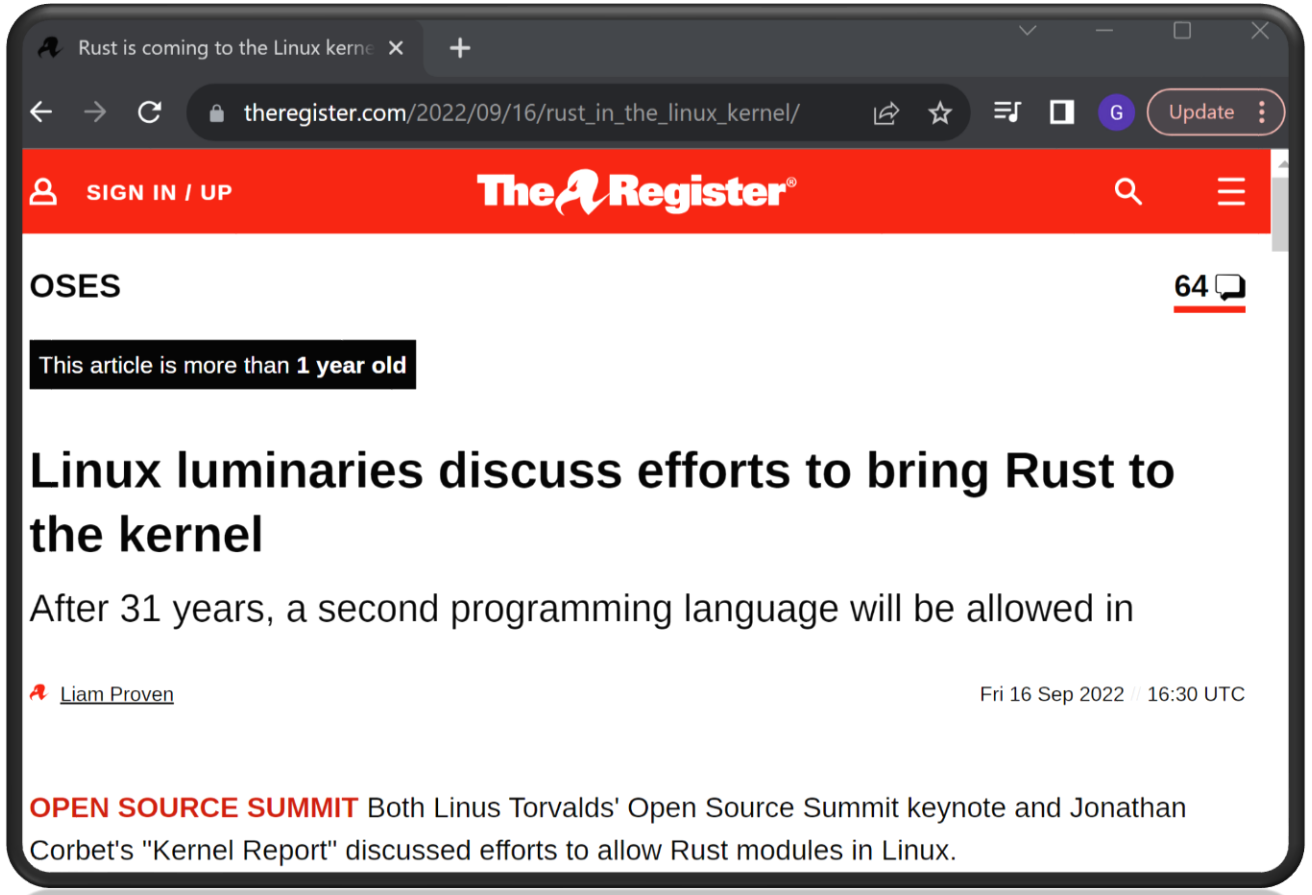
Rust nation UK (2024)

https://www.youtube.com/watch?v=6mZRWFQRvmw&t=27012s



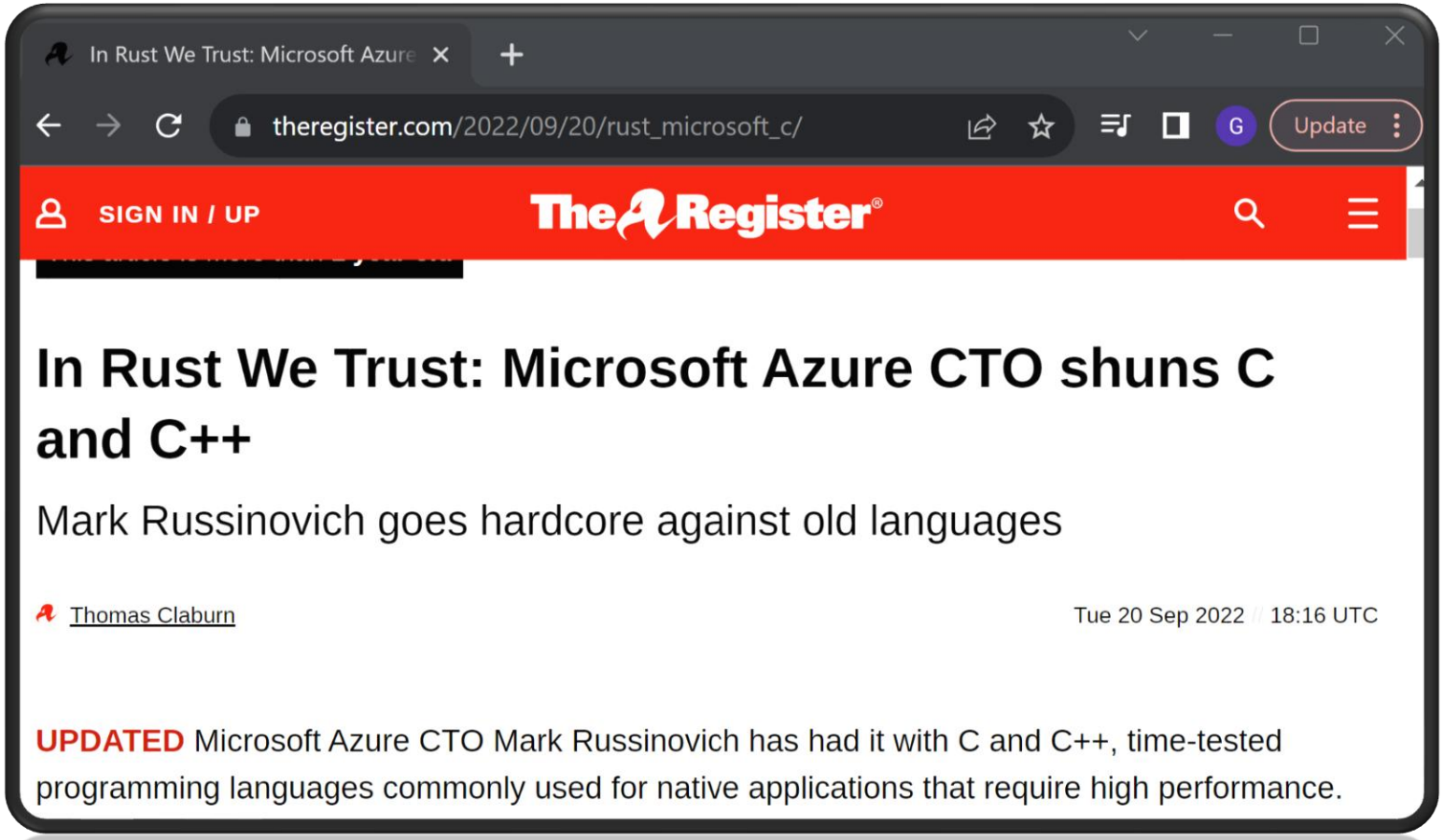
Rust History

- 2022/Sep → Rust for Linux Kernel is announced to be released in Linux kernel 6.1



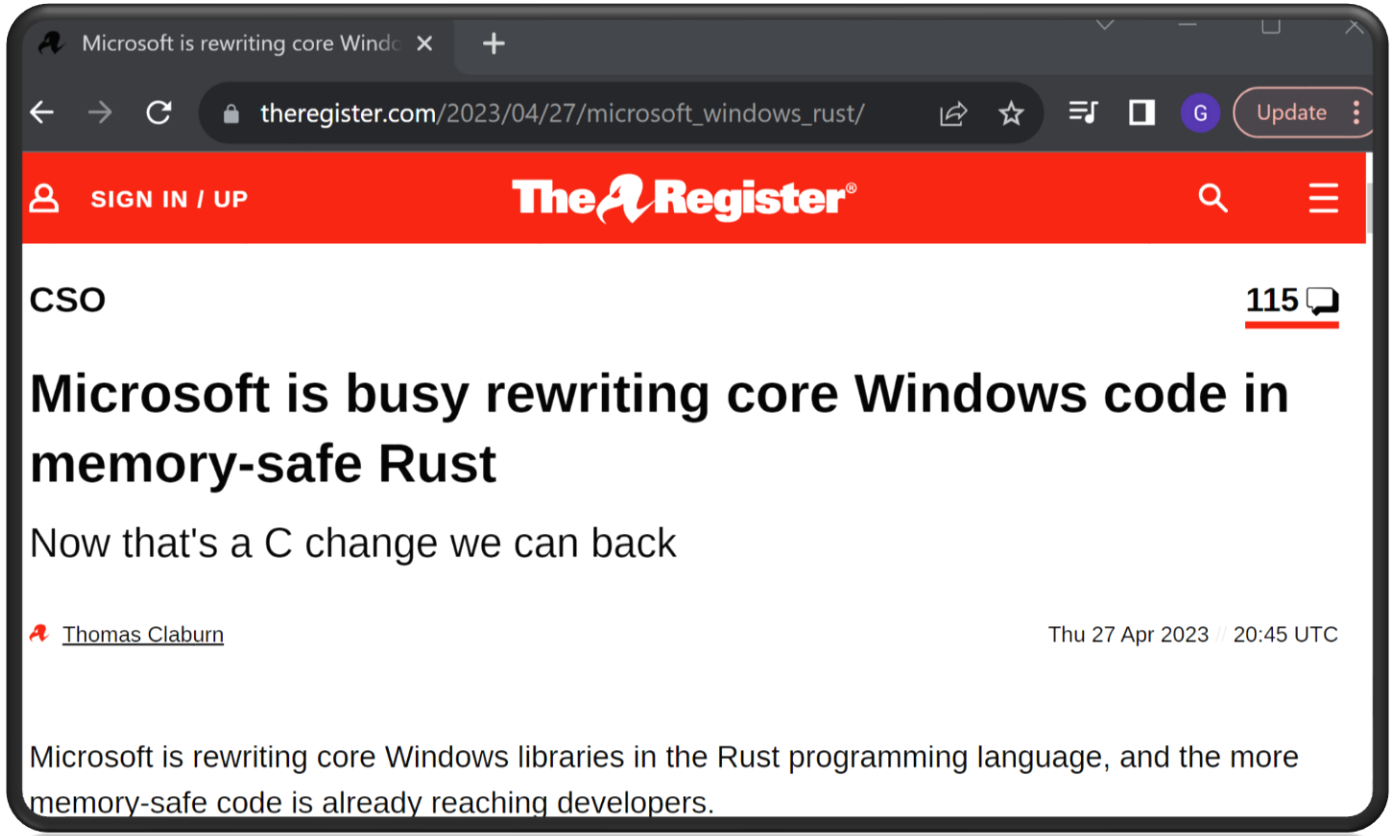
Rust History

- 2022/Sep → Azure announce its support for Rust programming



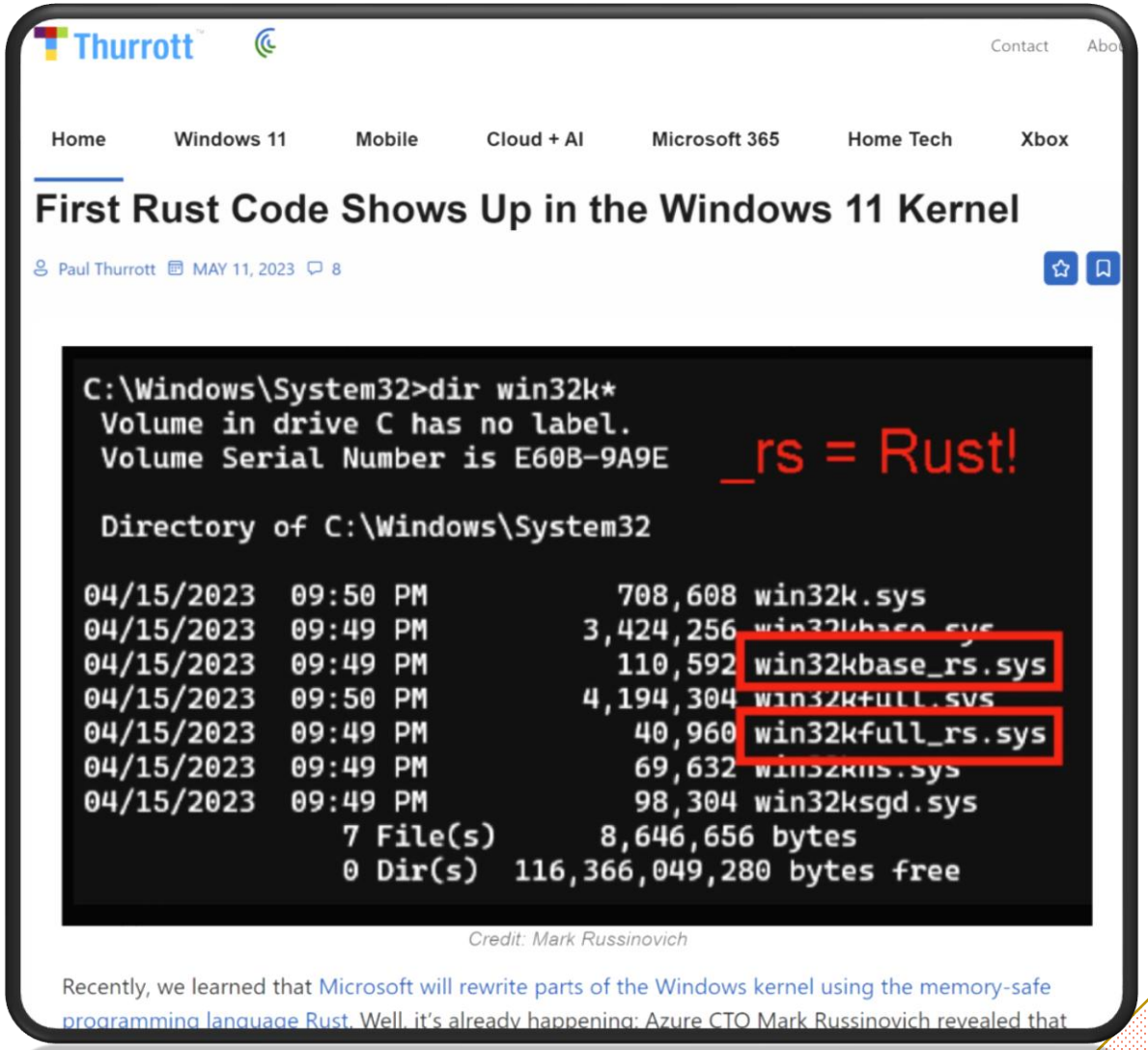
Rust History

- Close after that event, Microsoft started to change some of its internal code to Rust.



Rust History

- And after Microsoft Build Conference from 2023, Microsoft announces its first kernel components written in Rust as part of their ecosystem.



Rust History

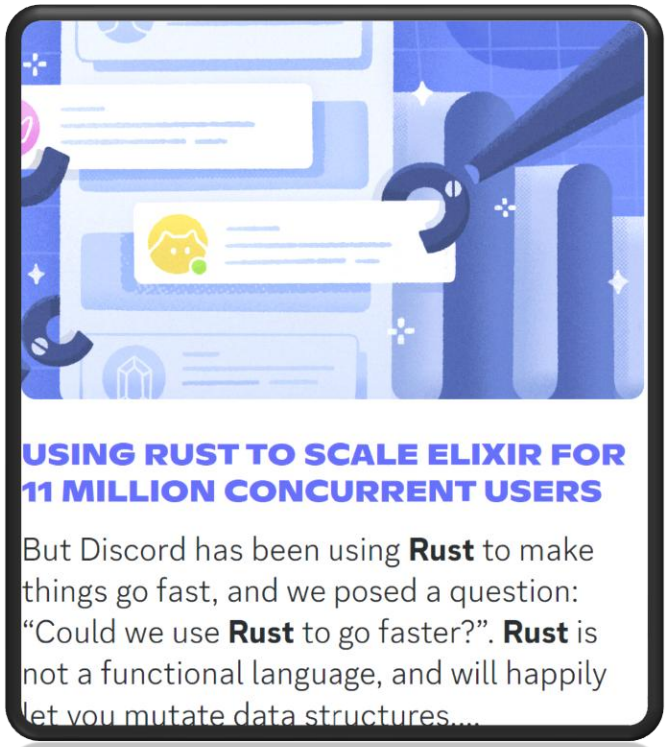
- In May.2024, Microsoft donates 1M USD to Rust Foundation to confirm company interest in this language.

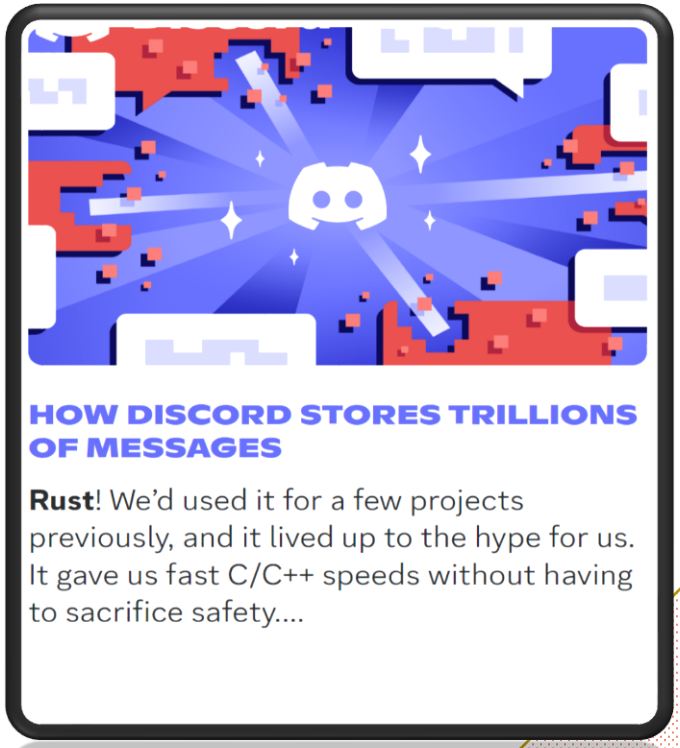
Rust History

- Additionally, NSA has issued a document that suggest using memory safety languages (such as Rust)

Rust History

- Finally, it is worth mention that Discord uses Rust on several backend projects that require memory safety and increase performance: https://discord.com/blog/search?query=rust





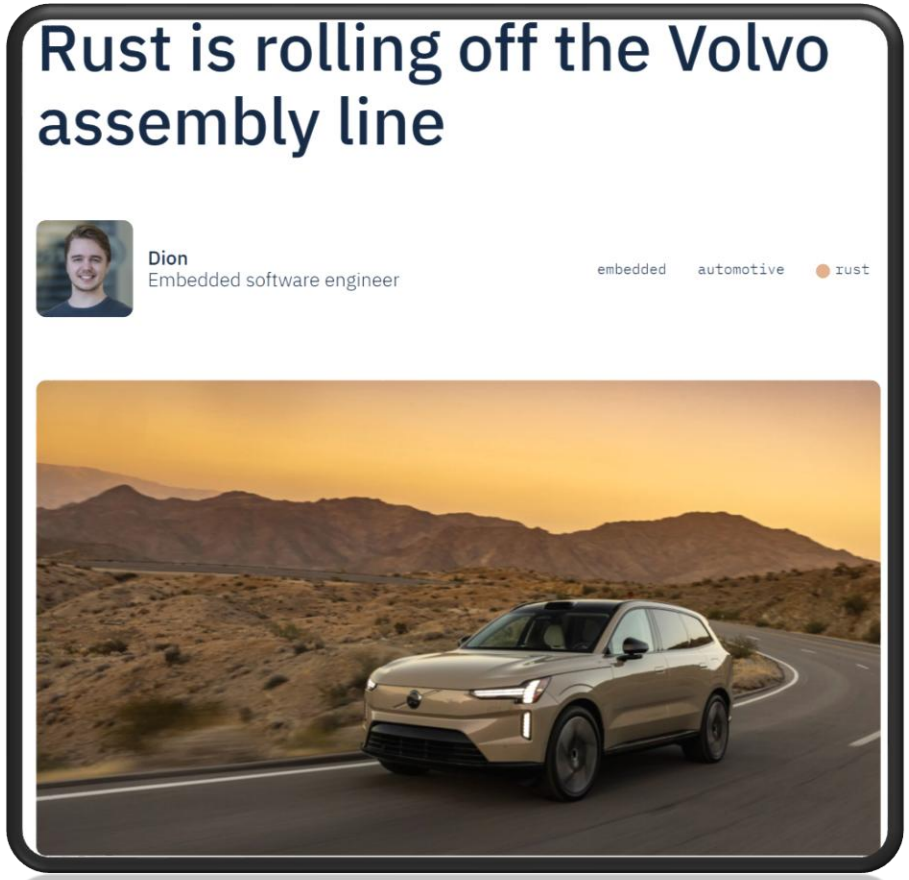
Rust History

Other memorable notions:

- **2022** → CloudFlare announced Pingore (their proxy that connects Cloudflare to Internet – written in Rust): https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/
- **2022** → Facebook announced their support for Rust for server side components: https://www.zdnet.com/article/the-rust-programming-language-just-got-a-big-boost-from-meta/
- **2022** → Google announced that they started to use Rust for Android to mitigate risks: https://www.zdnet.com/article/google-after-using-rust-we-slashed-android-memory-safety-vulnerabilities/
- **2023** → Github switch to a new search engine (BlackBird) written completely in Rust: https://www.zdnet.com/article/github-builds-a-search-engine-for-code-from-scratch-in-rust/
- **2023** → Meta announces Buck2 (a build system written in Rust): https://engineering.fb.com/2023/10/23/developer-tools/5-things-you-didnt-know-about-buck2/

Rust History

Or other a little bit different:



Rust IDEs

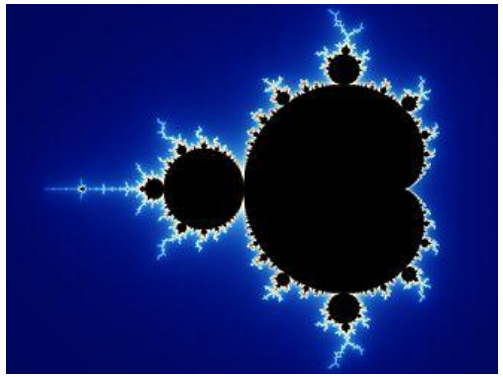
- **Visual Studio Code:**
  https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer
- **IntelliJ (RostOver):**
  https://blog.jetbrains.com/rust/2023/09/13/introducing-rustrover-a-standalone-rust-ide-by-jetbrains/
- **Eclipse:**
  https://www.eclipse.org/downloads/packages/release/2019-09/r/eclipse-ide-rust-developers-includes-incubating-components
- **Online IDE (Rust playground):**
  https://play.rust-lang.org/
- **Other online compilers:**
  https://www.tutorialspoint.com/compile_rust_online.php
  https://replit.com/languages/rust
  https://www.onlinegdb.com/online_rust_compiler
  https://rust.godbolt.org/

Rust Characteristics

- Strong-typed & statically typed language
- LLVM backend (native compiler) – gcc backend also a possibility in the future
- Ownership and lifetimes for variables
- Memory safety (allocation / access)
- No garbage collector
- Zero cost abstraction
- Move semantics
- Traits (for polymorphism)
- Package manager and build mechanisms

Performance tests (Mandelbrot)

The **Mandelbrot test** is a **benchmarking exercise** that measures the **performance of a programming language or compiler** by computing and rendering the **Mandelbrot** fractal.



What It Measures:

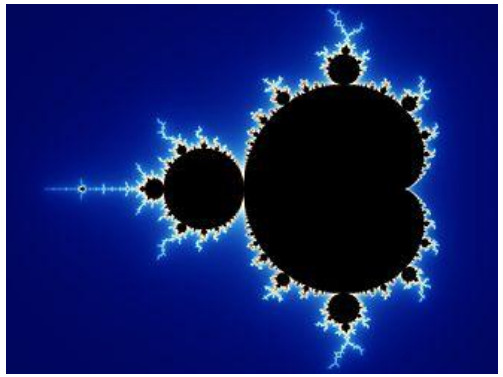
- Raw **CPU performance**
- **Loop optimization**
- **Floating-point performance**
- **Parallelism or threading** efficiency
- Compiler **code generation quality**

Language	Time	Peak Memory	Version
Rust	247ms	4.9MB	rustc 1.89.0-nightly
Rust	291ms	4.8MB	rustc 1.87.0
C	332ms	6.0MB	zigcc 0.14.1
C-Sharp	332ms	37.3MB	dotnet 9.0.300
C	452ms	6.5MB	clang 14.0.0-1ubuntu1.1
C	543ms	6.6MB	gcc 15.1.0
Nim	578ms	4.5MB	nim 2.2.4
Java	1162ms	55.7MB	openjdk 23
Java	1167ms	54.6MB	openjdk 21
Java	1194ms	108.9MB	graal/jvm 17.0.8
Go	3230ms	7.7MB	go 1.24.3

https://programming-language-benchmarks.vercel.app/problem/mandelbrot

Performance tests (Mandelbrot)

The **Mandelbrot test** is a **benchmarking exercise** that measures the **performance of a programming language or compiler** by computing and rendering the **Mandelbrot** fractal.



What It Measures:

- Raw **CPU performance**
- **Loop optimization**
- **Floating-point performance**
- **Parallelism or threading** efficiency
- Compiler **code generation quality**

#	Language	Time (sec)	Memory (MB)
1	**Rust**	0.95	35,598
2	**Chapel**	1.18	42,156
3	**Julia**	1.54	357,138
4	**C gcc**	1.64	35,582
5	**C++ g++**	2.36	38,281
6	**Intel Fortran**	2.72	85,975
7	**Go**	3.77	37,970
8	**Free Pascal**	3.91	35,529
9	**Java**	3.96	58,348
10	**Ada 2012**	4.01	41,099
11	**C#**	4.02	40,743
12	**Node.js**	4.05	144,757
13	**Lisp**	4.20	60,654

#	Language	Time (sec)	Memory (MB)
14	**Dart**	4.29	45,175
15	**Haskell**	6.64	51,057
16	**F#**	7.17	49,979
17	**Swift**	7.27	49,312
18	**OCaml**	7.60	64,643
19	**Erlang**	53.86	98,140
20	**PHP**	68.29	53,531
21	**Ruby**	143.13	118,436
22	**Lua**	159.01	652,796
23	**Python 3**	182.94	62,173
24	**PHP**	258.19	16,437
25	**Smalltalk**	>5 min	175,542
26	**Perl**	>8 min	114,569

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/mandelbrot.html

(code that have possible hand-written vector instructions or "unsafe“ was removed)

First rust program

First RUST Program

Rust

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

- C-like syntax

First RUST Program

Rust

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

C/C++

```c
void main()
{
    printf("Hello world !");
}
```

- C-like syntax
- However, there are some differences:
  - A function in C is defined by writing the return type first, while in Rust a function is defined using a special keyword fn
  - “printf” is a function in C/C++, while “print!” is a macro in Rust

First RUST Program

Rust

```rust
fn helloWorld() -> i32
{
    print!("Hello world !");
    return 0;
}
```

C/C++

```cpp
int helloWorld()
{
    printf("Hello world !");
    return 0;
}
```

- C-like syntax
- However, there are some differences:
  - A function in C is defined by writing the return type first, while in Rust a function is defined using a special keyword fn
  - “printf” is a function in C/C++, while “print!” is a macro in Rust
- To specify the return value of a function, use the following syntax: “-> <type>”

Create your very first RUST program

1. Using rustc (rust compiler) command line:

- Make sure that rust is installed
- Create a file in a folder named “first.rs” and insert into it the “hello world example (the one with a main function)
- Run the following command from command line: `rustc first.rs`
- An executable file (e.g. first.exe if you run this command in Windows) should appear in the first.rs file
- Run the executable file created on the precedent step (e.g. run first.exe if you are on Windows)

Create your very first RUST program

2. Using cargo (rust package manager) from command line:

- Make sure that rust is installed
- Run the following command from command line: cargo new first
- You should see a new folder (named first) that was created in the current folder with the following structure:

****	Current folder
\first	A folder that contains your first project
\first\.git	A hidden folder with a git integration data
\first\.gitignore	Ignore rules for git repo
\first\Cargo.toml	Configuration file for first project (INI like format)
\src	A folder with all rust sources
\src\main.rs	The main file of the rust project

Create your very first RUST program

2. Using cargo (rust package manager) from command line:

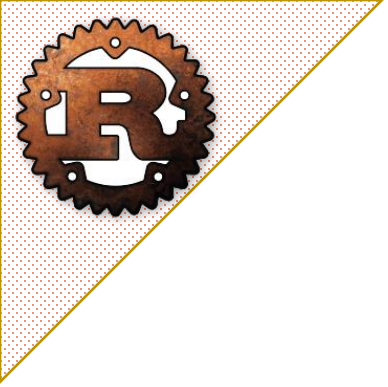
- Make sure that rust is installed
- Run the following command from command line: cargo new first
- You should see a new folder (named first) that was created in the current folder with the following structure:

\
\first
\first\.git
\first\.gitignore
\first\Cargo.toml
\src
\src\main.rs

```
[package]
name = "first"
version = "0.1.0"
edition = "2022"

# See more keys and their definitions at…

[dependencies]
```



Create your very first RUST program

2. Using cargo (rust package manager) from command line:

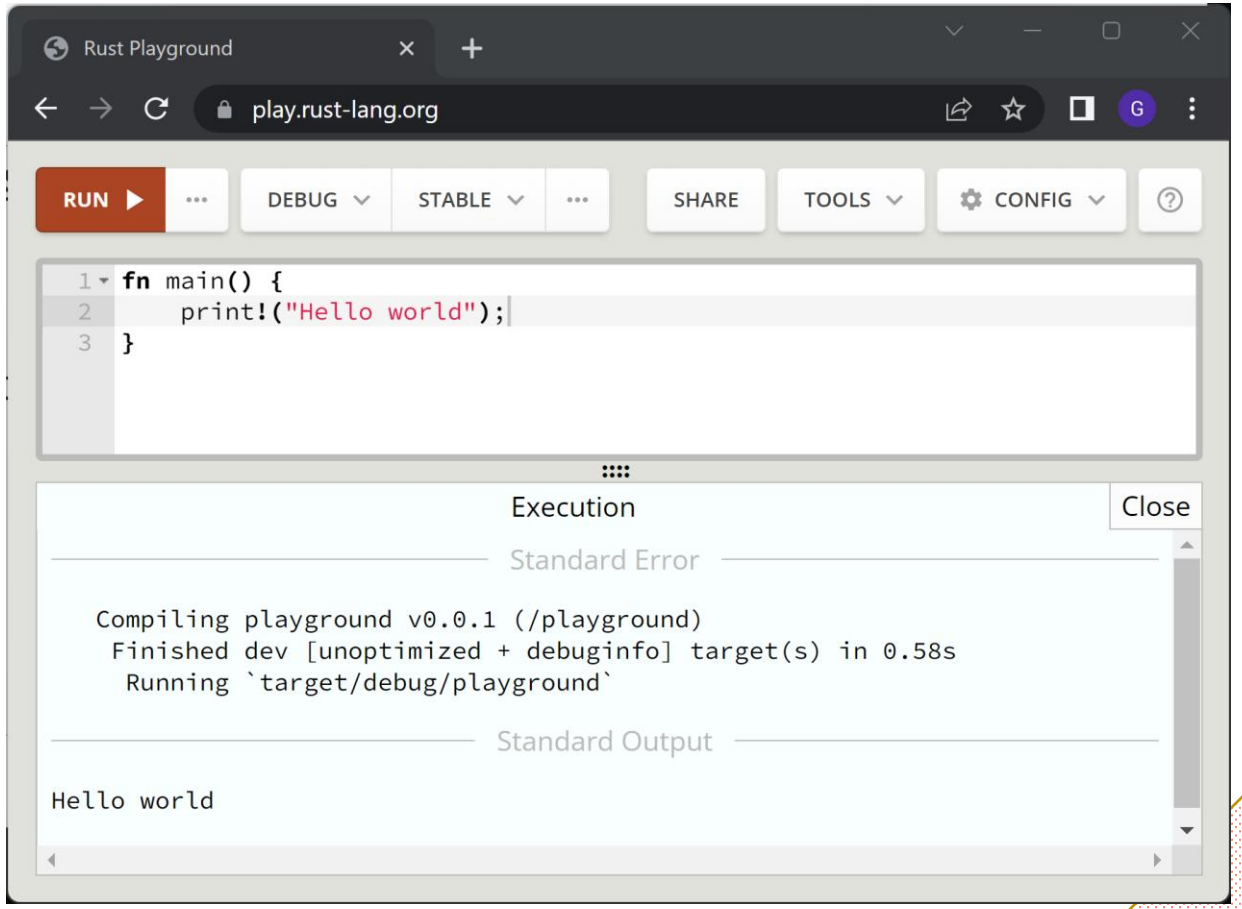
- Make sure that rust is installed
- Run the following command from command line: cargo new first
- You should see a new folder (named first) that was created in the current folder.
- Modify the "**\src\main.rs**" to contain the hello world example
- In folder "**\first**" execute the following command: cargo run

```
\first>cargo run
   Compiling first v0.1.0 (E:\Lucru\Rust\temp2\first)
    Finished dev [unoptimized + debuginfo] target(s) in 0.81s
     Running `target\debug\first.exe`
Hello, world!
```

Create your very first RUST program

3. Try it using rust playground:

- Open a browser and go to https://play.rust-lang.org/
- Write the hello world code
- Hit the Run button from the top-left side of the web-page



Create your very first RUST program

4. Use different features from specialized IDEs:

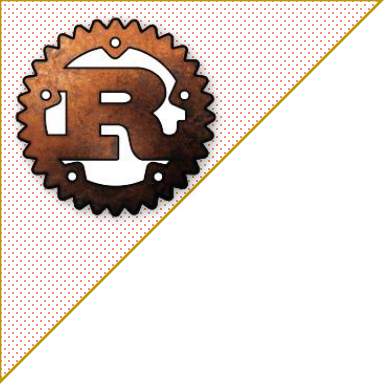
- Use features such as create new project (IntelliJ) or various command/prompts from Visual Studio Code
- In the backend the **cargo** utility is usually used

Basic Types

Basic types

Rust has several basic types, including:

1. Integers
2. Float values
3. Boolean
4. Character type



Basic types

Rust integer types:

Size	Rust type	C/C++ type	C/C++ (approximate type)
8 bit (unsigned)	u8	uint8_t	unsigned char
8 bit (signed)	i8	int8_t	char
16 bit (unsigned)	u16	uint16_t	unsigned short
16 bit (signed)	i16	int16_t	short
32 bit (unsigned)	u32	uint32_t	unsigned int
32 bit (signed)	i32	int32_t	int
64 bit (unsigned)	u64	uint64_t	unsigned long long
64 bit (signed)	i64	int64_t	long long
128 bit (unsigned)	u128	N/A*	N/A
128 bit (signed)	i128	N/A*	N/A

* gcc and c-lang provide a __int128 type (but it is not part of the standard)

Basic types

Rust integer types:

- General format is “i<no of bits>” for signed integer or “u<no of bits>” for unsigned integers
- Signed integers (just like in C) are based on C_2 complement format
- Besides this, there are two more integers types that have variable length (depending on the architecture (32 or 64 bit). These types are used as indexes in an array or as a representation of an offset or size of a structure in memory.

Type	Rust type	C/C++ type
Unsigned (32/64 bit)	usize	size_t
Signed (32/64 bit)	isize	ptrdiff_t

Basic types

Rust integer types:

- Rust supports the following notations for integer values:
  - 0x => for hexadecimal values
  - 0o => for octal values
  - 0b => for binary values
  - b'char' => for u8 values
- In addition, rust supports the use of character _ as a digit delimitator
- The type of the integer value (u8,u16,i8,i16,u32 …) can also be specified as a suffix for a number to specify its type

Basic types

Rust integer types:

Rust

```rust
123;           // i32
0x12u8;        // u8
0b11001100i64; // i64 value
123_456_789;   // i32 value of 123456789
0xFF_FF_FF_FF; // i32 value with delimitators
b'A';          // an u8 with value 65 (ascii code for character 'A')
0o11_33_77u64; // u64 value written in octal mode with delimitators
100usize;      // value 100 as type usize;
```

Basic types

Rust float types:

- General format is “f<no of bits>” where number of bits is either 32 or 64
- Format IEEE-754 (similar to C/C++)
- “f32” and “f64” can also be used as suffixes when creating a float constant

Type	Rust type	C/C++ type
Float 32 bits	f32	float
Float 64 bits	f64	double

Rust

```
123.0;          // f64
123f64;         // 123.0 (f64)
1.23f32;        // f32 value
5.200_345;      // f64 value of 5.200345
```

Basic types

Rust boolean type:

- Identical to C/C++ (bool)

Type	Rust type	C/C++ type
Boolean	bool	bool

- Size of bool type is 1 byte (just like in C/C++) with 1 indicating a true value and 0 a false value
- Same constants just like in C/C++: true and false

Basic types

Rust character type:

- Identical to C/C++ (char)

Type	Rust type	C/C++ type (used by people)	C++ equivalent type
Character	char	char	char32_t

- However, while the size of one character (for type ***char***) in C/C++ is one byte, in Rust the size of one character is 4 bytes (so that it can represent any Unicode character value). The equivalent type in C++ is ***char32_t***
- Character constants can be written using single quotes ‘A’

Variables & constants

Variable & Constants

Rust supports both local and global variables.

A local variable in Rust is declared using the special keywork let .

let *<variable_name>* : *<type>* **=** *<initialization_value>*;	
let *<variable_name>* **=** *<initialization_value>*;	Variable type is inferred from the *initialization_value*
let *<variable_name>* **:** *<type>* ;	Uninitialized variable. Its value must be set up before using it.
let *<variable_name>*	Uninitiatlize variable without a type specification. Its value must be set up before using it. When its value is going to be set up, its type is going to be inferred at that time.

Variable & Constants

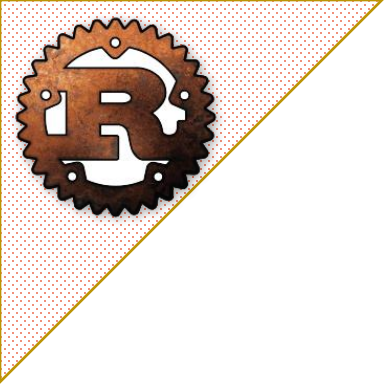
A simple example of local variables in Rust.

Rust

```rust
fn main() {
    let a: u32 = 123;    // a is of type u32 and has a value of 123
    let b = 123;         // b is inferred as an i32 and has a value of 123
    let c = 0xFFu64;     // c is of type u64 (due to the suffix of u64 from the
                         // initialization constant)
    let d: bool;         // d is of type bool and must be initialized before using
    let e;               // e is not initialized and its type will be inferred upon
                         // initialization.
    d = true;            // d value is initialized
    e = 4.3;             // e is initialized and its type is f64
    println!("a={}, b={}, c={}, d={}, e={}", a, b, c, d, e);
}
```

Output

```
a=123, b=123, c=255, d=true, e=4.3
```



Variable & Constants

Rust does not allow the usage of uninitialized variables:

Rust

```rust
fn main() {
    let x: i32;
    println!("x={}", x);
}
```

Error

```
error[E0381]: used binding `x` isn't initialized
 --> src/main.rs:3:22
  |
2 |     let x: i32;
  |         - binding declared here but left uninitialized
3 |     println!("x={}", x);
  |                      ^ `x` used here but it isn't initialized
```

Variable & Constants

However, the following code will work:

Rust

```rust
fn main() {
    let x: i32;
    println!("Before x is initialized !");
    x = 100;
    println!("x={}", x);
}
```

Output

```
Before x is initialized !
x=100
```

In the second case the type of “x” is inferred to i32.

Rust

```rust
fn main() {
    let x;
    println!("Before x is initialized !");
    x = 100;
    println!("x={}", x);
}
```

Variable & Constants

By default, any variable declared in Rust is immutable. The following code will not compile because variable “x” can not be modified.

Rust

```rust
fn main() {
    let x = 10;
    println!("x is {}", x);
    x = 100;
    println!("now x is {}", x);
}
```

Error

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src\main.rs:7:5
  |
5 |     let x = 10;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
6 |     println!("x is {}",x);
7 |     x = 100;
  |     ^^^^^^^ cannot assign twice to immutable variable
```

The usage of an in-mutable variable can be optimized when compiling (e.g inline-ing variable value).

Variable & Constants

In contrast, any C/C++ variables are by default mutable and can be changed. As such, declaring a variable in Rust is similar to the following code in C/C++;

Rust

```rust
fn main() {
    let x = 10;
    println!("x is {}", x);
}
```

C/C++ (using const specifier)

```c
void main() {
    const int x = 10;
    printf("x is %d",x);
}
```

C/C++ (using constexpr specifier)

```c
void main() {
    constexpr int x = 10;
    printf("x is %d",x);
}
```

Variable & Constants

To declare a mutable variable, use the keyword mut in the following way: let mut <variable_name> ….

Rust

```rust
fn main() {
    let mut x = 10;
    println!("x is {}", x);
    x = 100;
    println!("now x is {}", x);
}
```

Now the previous code compiles and produces the following output:

Output

```
x is 10
now x is 100
```

Variable & Constants

Once a variable has a type assigned (via initialization or through type inference) its type **CAN NOT** be changed (similar to how C/C++ works).

Rust

```
fn main() {
    let mut x;
    x = 10;
    println!("x is {}", x);
    x = true;
    println!("now x is {}", x);
}
```

Error

```
error[E0308]: mismatched types
 --> src\main.rs:8:10
  |
5 |     let mut x;
  |         ----- expected due to the type of this binding
...
8 |     x = true;
  |         ^^^^ expected integer, found `bool`
```

In this case the second assignment tries to set the value of variable “x” that is of type “i32” to bool (via constant true)

Variable & Constants

Keep in mind that the same code in C/C++ works (due to the rules of promotion, any Boolean value can be converted into an int).

Rust

```
fn main() {
    let mut x;
    x = 10;
    println!("x is {}", x);
    x = true;
    println!("now x is {}", x);
}
```

Compile error

C/C++

```
void main() {
    int x;
    x = 10;
    printf("x is %d\n",x);
    x = true;
    printf("now x is %d\n",x);
}
```

Compiles ok

Output

```
x is 10
now x is 1
```

Variable & Constants

As a general observation, once a variable is declared with a specific type, it can not be assigned with a value of a different type, even if an implicit conversion is possible.

Rust

```rust
fn main() {
    let mut x = 10u32;
    x = 1u8;
    println!("x is {}", x);
}
```

Error

```
error[E0308]: mismatched types
 --> src\main.rs:6:10
  |
5 |     let mut x = 10u32;
  |                 ----- expected due to this value
6 |     x = 1u8;
  |         ^^^ expected `u32`, found `u8`
```

In this case “X” is of type u32 (with values between 0 and 2^{32}-1) and the assigned value is of type u8. Even if every possible value for an u8 (values between 0 and 2^{32}-1) can be stored in an u32 value, Rust will still provide a compiler error.

Variable & Constants

In contrast, the same C/C++ code compiles (rules of conversion will automatically convert a u8 value to u32).

Rust

```rust
fn main() {
    let mut x = 10u32;
    x = 1u8;
    println!("x is {}", x);
}
```

Compile error

C/C++

```cpp
void main() {
    unsigned int x = 10;
    x = (unsigned char)1;
    printf("x is %d\n",x);
}
```

Compiles ok

Output

```
x is 1
```

Variable & Constants

To cast from a type to another type in Rust there is a keyword called as that can be used. The general format is <value> as <newType> :

For the previous case to work we need to rewrite it like this:

Rust

```rust
fn main() {
    let mut x = 10u32;
    x = 1u8 as u32;
    println!("x is {}", x);
}
```

Output
x is 1

Variable & Constants

Let’s see some examples:

Rust

```rust
fn main() {
    let mut x = 255 as u8;
    let mut y = 1 as i32;
    y = x as i32;
    println!("y is {}", y); // y is 255
}
```

Output

```
y is 255
```

Rust

```rust
fn main() {
    let mut x = 255 as u8;
    let mut y = 1 as i8;
    y = x as i8;
    println!("y is {}", y); // y is -1
}
```

Output

```
y is -1
```

First case is a simple one (as all possible values from u8) can be stored in an i32. The second case (i8 to u8) keeps the value as it is (in terms of bits) and just interprets the value in a different way.

Variable & Constants

Not all casts are possible (for example a cast between a number and a bool is not allowed). The following code will not compile:

Rust

```rust
fn main() {
    let mut x = 1 as u64;
    let mut y = true;
    y = x as bool;
    println!("y is {}", y);
}
```

Error

```
error[E0054]: cannot cast as `bool`
 --> src\main.rs:7:10
  |
7 |      y = x as bool;
  |          ^^^^^^^^^ help: compare with zero instead: `x != 0`
```

C/C++ has a different logic (any value that is 0 can be implicitly seen as false and any value different than 0 can implicitly be seen as true).

Variable & Constants

Let’s see some examples:

Rust

```rust
fn main() {
    let mut x = 258 as u64;
    let mut y = 0 as u8;
    y = x as u8;
    println!("y is {}", y); // y is 2
}
```

Output
y is 2

In this case, a truncation to the first 8-bits happens for the value of 258 (keep in mind that an u8 cand only store values between 0 and 255).

258 = 0000 0000 0000 0000 0000 0001 0000 0010

= the code will truncate to the last 8bits (least significant one)

= as such the result of the cast will be 1

Variable & Constants

In case of integer to float conversions, the program attempts to obtain the closest float value to the integer one (that can be represented in the float format).

Rust

```rust
fn main() {
    let x = 100 as u32;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}",y);
    println!("x is {}",x);
}
```

Output

```
y is 100
x is 100
```

Rust

```rust
fn main() {
    let x = 0xFFFF_FFFF as u32;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}",y);
    println!("x is {}",x);
}
```

Output

```
y is 4294967300
x is 4294967295
```

Notice the difference

Variable & Constants

In case of float to float conversion (every f32 can be converted into an f64, while an f64 is approximated to the closest f32).

Rust

```rust
fn main() {
    let x = 1.0123456789 as f64;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Output

```
y is 1.0123457
x is 1.0123456789
```

Rust

```rust
fn main() {
    let x = 1.79769313486231570E+308 as f64;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Output

```
y is inf
x is 179769313486231570000....000
```

If the f64 value is outside maximum value possible in f32, the conversion will enforce inf as the value of f32.

Variable & Constants

To declare multiple variables at the same time, use the following format:

- let (var_1, var_2, … var_n) = ($value_1$, $value_2$, … $value_n$)

The types for var_1, var_2, … var_n are inferred from the associated values.

Rust

```rust
fn main() {
    let (mut x, mut y, mut z) = (1, 2, 3);
    let (a, b) = (1.23, true);
    println!("{x},{y},{z},{a},{b}");
    x = 10;
    y = 20;
    z = 30;
    println!("{x},{y},{z},{a},{b}");
}
```

Output
1,2,3,1.23,true 10,20,30,1.23,true

The number of values provided must be the same as the number of variables.

Variable & Constants

Rust also allows **shadowing** → meaning that a variable with the same name and possible different type can be declared within an inner block and possible be initialized with the value of the initial variable.

Rust

```rust
fn main() {
    let x: i32 = 10;
    println!("x is {}", x);
    {
        let x: bool = true;
        println!("inner x is {}", x);
    }
    println!("outter x is {}", x);
}
```

Output

```
x is 10
inner x is true
outter x is 10
```

Variable & Constants

Shadowing is often used to change the mutability state of one variable for a limited period of time by using a copy of that variable.

Rust

```rust
fn main() {
    let x: i32 = 10;
    println!("x is {}", x);
    {
        let mut x = x;
        x = 20; // x is now mutable and can be changed
        println!("inner x is {}", x);
    }
    println!("outter x is {}", x);
}
```

Output

```
x is 10
inner x is 20
outter x is 10
```

Variable & Constants

For global variables, use the keyword static instead of let (let is designed for stack/local variables).

Rust

```rust
static x: i32 = 10;

fn main() {
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Output

```
y is 10
x is 10
```

A static variable implies a possible mutable variable (if mut keyword is being used) and guarantees an allocated space in the binary data.

Variable & Constants

Rust also allows the usage of the special keyword const to define a constant. The main difference between a ***const value*** and a ***static value*** is that a ***const value*** is always immutable and will be replaced with its value upon compiling phase.

Rust

```rust
const x: i32 = 10;

fn main() {
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Output

```
y is 10
x is 10
```

Variable & Constants

Both static and const keywords can be used in a local function:

Rust

```rust
fn main() {
    static x: i32 = 10;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Rust

```rust
fn main() {
    const x: i32 = 10;
    let mut y = 0 as f32;
    y = x as f32;
    println!("y is {}", y);
    println!("x is {}", x);
}
```

Output

```
y is 10
x is 10
```

OBS: a mutable static value requires unsafe code (more about this on a different course).

Variable & Constants

Another difference between static / const and let is that *type* has to be provided in case of static and const keywords and can not be inferred.

Rust

```
fn main() {
  const x = 10;
  println!("x is {}",x);
}
```

Error

```
error: missing type for `const` item
 --> src\main.rs:4:12
  |
4 |      const x = 10;
  |            ^ help: provide a type for the constant: `x: i32`
```

Rust

```
fn main() {
  static x = 10;
  println!("x is {}",x);
}
```

Error

```
error: missing type for `const` item
 --> src\main.rs:4:12
  |
4 |      const x = 10;
  |            ^ help: provide a type for the static variable: `x: i32`
```

Variable & Constants

The same logic applies for initialization value (if in case of let, one can define a variable and assigned its value after this), in case of const and static this is not possible.

Rust

```rust
fn main() {
  static x: i32;
  x = 10;
  println!("x={}",x);
}
```

Error

```
error: free static item without body
 --> src\main.rs:4:6
  |
4 |      static x: i32;
  |      ^^^^^^^^^^^^^-
  |                   |
  |                   help: provide a definition for the static: `= <expr>;`
```

Variable & Constants

Observations:

1. There is no such thing as 0 initialization in Rust (for global variables).
   From this point of view, Rust tries to make everything clear and make sure that there is no unknown behavior due to this case.
2. In case of constants, Rust recommend using upper cases (this is considered a warning and will not affect the compilation phase).

Warning

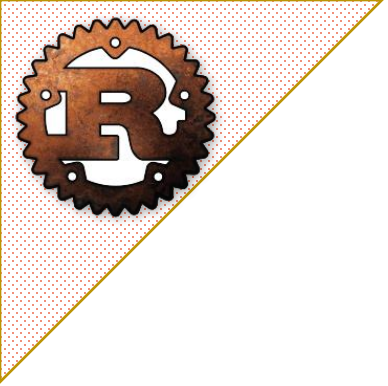
```
warning: constant `x` should have an upper case name
 --> src\main.rs:4:12
  |
4 |      const x: i32 = 10;
  |            ^ help: convert the identifier to upper case (notice the capitalization): `X`
  |
  = note: `#[warn(non_upper_case_globals)]` on by default
```

Variable & Constants

Comparation with C/C++ similar forms:

Keyword	Rust	C/C++
let	let mut x: i32 = 10;	int x = 10;
	let mut x: i32;	int x;
	let mut x = 10;	auto x = 10;
	let mut x; x = 10;	N/A
const	const x: i32 = 10;	const int x = 10;
		constexpr int x = 10;
		#define x 10
static	static x: i32 = 10;	const int x = 10; // as a global variable
		static const int x = 10; // as a local definition

Operators



Operators

Rust supports the following operators:

Operator	Rust	C/C++
Arithmetic (+ - * / %)	Yes	Yes
Comparation (> < >= <= != ==)	Yes	Yes
Logical OR , AND or NOT (|| && !)	Yes	Yes
Bit operators: bitwise OR, bitwise AND, shifts (& | ^ >> <<)	Yes	Yes
Assignments (= += -= *= %= /= &= |= ^= >>= <<=)	Yes	Yes
Increment / Decrement operators (++ --)	N/A	Yes
Negate operator (~)	N/A (use !)	Yes
Conditionally operator (?:) → **condition ? Value for true : Value for false**	N/A	Yes
Member access operator (. ->)	Partial (only .)	Yes
Error propagation (?)	Yes	N/A
Range literals (.. ..= ...)	Yes	N/A
Functional update (..)	Yes	Partial (proxy ctor)

Operators

Rust has a very comprehensive compiling error system that besides clearly explaining an error, it also provides some directions/ideas on how to fix that error. In particular for operators, Rust can identify some of the widely used operators that are not supported (such as increment or decrement) and provide guidance on how to fix some errors.

Rust

```
fn main() {
    let mut x = 10;
    x++;
    print("x = {}", x);
}
```

Error

```
error: Rust has no postfix increment operator
 --> src\main.rs:5:7
  |
5 |      x++;
  |       ^^ not a valid postfix operator
  |
help: use `+= 1` instead
  |
5 |      { let tmp = x; x += 1; tmp };
  |      ++++++++++  ~~~~~~~~~~~~~~~
5 -      x++;
5 +      x += 1;
```

Operators

Rust has a very comprehensive compiling error system that besides clearly explaining an error, it also provides some directions/ideas on how to fix that error. In particular for operators, Rust can identify some of the widely used operators that are not supported (such as increment or decrement) and provide guidance on how to fix some errors.

Rust

```
fn main() {
    let mut x:u32 = 10;
    x = ~x;
    print!("x = {}", x);
}
```

Error

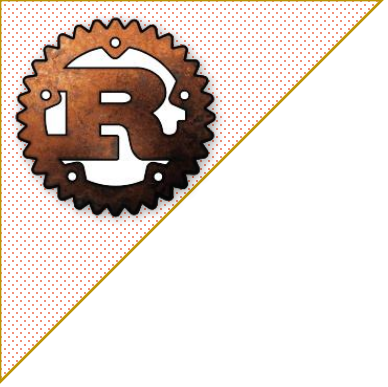
```
error: `~` cannot be used as a unary operator
 --> src\main.rs:5:10
  |
5 |      x = ~x;
  |          ^ help: use `!` to perform bitwise not
```

Operators

Operators order:

Operator	Associativity
* / %	left to right
+ -	left to right
<< >>	left to right
&	left to right
^	left to right
|	left to right
== != < > <= >=	Require parentheses
&&	left to right
||	left to right
.. ..=	Require parentheses
= += -= *= /= %= &= |= ^= <<= >>=	right to left

Functions & Expression Statements



Functions

Function in Rust are defined using the keywork fn in the following way:

Rust	C/C++
fn <name> () {…}	void <name> () {…}
fn <name> (parameters) {…}	void <name> (parameters) {…}
fn <name> () -> <return_type> {…}	<return_type> <name> () {…}
fn <name> (parameters) -> <return_type> {…}	<return_type> <name> (parameters) {…}

Where parameters are defined in the following way:

* <param_name>:<type> [, < param_name >:<type>, …]
* mut < param_name >:<type> [, ….]

Where *<param_name>* is the name of a parameter, and <type> is the type of that variable.

Functions

Examples:

Rust

```rust
fn sum(x: u32, y: u32) -> u32 {
    return x + y;
}
fn print_x(x: u32) {
    println!("X is {}", x);
}
fn main() {
    let mut x: u32 = 10;
    print_x(x);
    x = sum(10, 20);
    print_x(x);
}
```

Output

```
X is 10
X is 30
```

Functions

Examples:

Rust

```rust
fn compute(x: u32, y: u32) -> u32
{
    x = x * y;
    return x + y;
}
fn print_x(x: u32) {
    println!("X is {}", x);
}
fn main() {
    let mut x: u32 = 10;
    print_x(x);
    x = compute(10, 20);
    print_x(x);
}
```

Error

```
error[E0384]: cannot assign to immutable argument `x`
 --> src\main.rs:4:5
  |
3 | fn compute(x:u32, y:u32) -> u32 {
  |            - help: consider making this binding mutable: `mut x`
4 |     x = x * y;
  |     ^^^^^^^^^ cannot assign to immutable argument
```

Functions

Examples:

Rust

```
fn compute(x:u32, y:u32) -> u32 {
    x = x * y;
    return x+y;
}
fn print_x(x:u32) {
    println!("X is {}",x);
}
fn main() {
     let mut x:u32 = 10;
     print_x(x);
     x = compute(10,20);
     print_x(x);
}
```

Rust

```
fn compute(mut x:u32, y:u32) -> u32 {
    x = x * y;
    return x+y;
}
fn print_x(x:u32) {
    println!("X is {}",x);
}
fn main() {
     let mut x:u32 = 10;
     print_x(x);
     x = compute(10,20);
     print_x(x);
}
```

Functions

Rust has a keyword (**return**) that can be used to return a value from a function (similar to C-like languages). At the same time, Rust allows a different format of returning a value by writing the value that you want to return directly.

The following two Rust programs are equivalent.

Rust

```rust
fn value_3() -> u32
{
    return 3;
}
fn main() {
    let x = value_3();
    println!("x is {}", x);
}
```

Rust

```rust
fn value_3() -> u32
{
    3
}
fn main() {
    let x = value_3();
    println!("x is {}", x);
}
```

Functions

Rust has a keyword (**return**) that can be used to return a value from a function (similar to C-like languages). At the same time, Rust allows a different format of returning a value by writing the value that you want to return directly.

The following two Rust pro

Notice that ; character is not added after the value that needs to be returned

Rust

```rust
fn value_3() -> u32
{
    return 3;
}
fn main() {
    let x = value_3();
    println!("x is {}", x);
}
```

```rust
fn value_3() -> u32
{
    3
}
fn main() {
    let x = value_3();
    println!("x is {}", x);
}
```

Expression statements

Rust also allows an expression statement where something gets computed based within a statement {…}

Rust

```rust
fn main() {
    let y = 10;
    let x = { 1 + y };
    println!("x = {x}");
}
```

Output
X = 11

Notice that the same return format (with a value) is being used in this case (without the ; character at the end)

Expression statements

In this format, some even more complex operations can be performed:

Rust

```rust
fn main() {
    let y = 10;
    let x = {
        let temp = y;
        let a = temp * y;
        a + y + 5
    };
    println!("x = {x}");
}
```

Output
X = 115

In this case, the value of “x” will be “a+y+5”

Expression statements

In this format, some even more complex operations can be performed:

Rust

```rust
fn main() {
    let y = 10;
    let x = {
        fn sum(x: i32, y: i32) -> i32 { x + y }
        fn dif(x: i32, y: i32) -> i32 { x - y }
        sum(2, 5) + dif(7, 3)
    };
    println!("x = {x}");
}
```

Output
X = 11

It is also possible to create nested functions within an expression statements and use them to compute the return value.

Expression statements

Nested functions can be added within an existing function:

Rust

```rust
fn main() {
    fn sum(x: i32, y: i32) -> i32 {
        x + y
    }
    fn dif(x: i32, y: i32) -> i32 {
        x - y
    }
    let y = sum(10, 20);
    let x = dif(20, 10);
    println!("x = {x}, y = {y}");
}
```

Output
X = 10, Y = 30

Basic statements

Basic statements

Most of basic statements that exists in C/C++ can be found in Rust as well:

Statement type	Rust	C/C++
If statement	Yes	Yes
While statement	Yes	Yes
For statement (classic)	N/A	Yes
For each	Yes	Yes
Loop	Yes	N/A
Do…While statement	N/A	Yes
GoTo	N/A (partial support)	Yes
switch	Yes	Yes
Patterns: if..let, while..let, let..else	Yes	N/A

if

If statement format:

- if condition <then statement>
- if condition <then statement> else <else statement>

Obs:

1. Notice that condition does not require parentheses (…)
2. Because of this, *<then statement>* and *<else statement>* can not be simple instructions (they have to be embedded in a block).

if

Example:

Rust

```rust
fn main() {
    let mut x = 1;
    if x > 0 {
        x += 1;
    }
    println!("x = {x}");
}
```

C/C++ (v1)

```c
void main() {
    int x = 1;
    if (x>0) {
        x+=1;
    }
    printf("x = %d", x);
}
```

C/C++ (v2)

```c
void main() {
    int x = 1;
    if (x>0)
        x+=1;
    printf("x = %d", x);
}
```

Output

```
X = 2
```

if

Example (if … else) :

Rust

```rust
fn main() {
    let mut x = 1;
    if x > 0 {
        x += 1;
    } else {
        x -= 1;
    }
    println!("x = {x}");
}
```

C/C++ (v1)

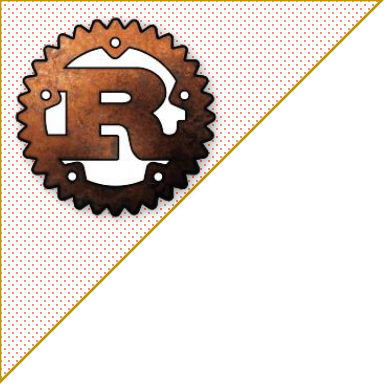
```c
void main() {
    int x = 1;
    if (x>0) {
        x+=1;
    } else {
        x-=1;
    }
    printf("x = %d", x);
}
```

C/C++ (v2)

```c
void main() {
    int x = 1;
    if (x>0)
        x+=1;
    else
        x-=1;
    printf("x = %d", x);
}
```

Output

```
X = 2
```



if

Parentheses (and) around the condition are allowed, the code compiles but triggers a warning:

Rust

```rust
fn main() {
    let mut x = 1;
    if (x > 0) {
        x += 1;
    }
    println!("x = {x}");
}
```

Warning

```
warning: unnecessary parentheses around `if` condition
 --> src\main.rs:5:8
  |
5 |     if (x>0) {
  |        ^   ^
  |
  = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
  |
5 -     if (x>0) {
5 +     if x>0 {
```

Output

```
X = 2
```

if

If statement can also be used as an expression statement:

Rust

```rust
fn main() {
    let x = 31;
    let y = if x > 20 { x / 2 } else { x * 2 };
    println!("y = {y}");
}
```

C++

```cpp
void main() {
    int x = 31;
    int y = x>20 ? x/2 : x * 2;
    printf("y = %d", y);
}
```

Output

```
y = 15
```

Notice the fact the return value is specified just like in the case of expression statements for both ***then*** and ***else*** parts.

while

While statement is similar to the one from C/C++:

- *while <condition> { … do block … }*

Just like in if statement case , notice that the ***<condition>*** does not need to be surrounded by parentheses.

Rust

```rust
fn main() {
    let mut x = 0;
    while x < 3 {
        println!("x={x}");
        x = x + 1;
    }
}
```

C/C++

```cpp
void main() {
    int x = 1;
    while (x<3) {
        printf("x = %d", x);
        x+=1;
    }
}
```

Output

```
X = 0
X = 1
X = 2
```

while

Similar to if statement , the condition MUST be followed by a block (and can not be a simple instruction like in the case of C/C++).

Rust

```rust
fn main() {
    let mut x = 0;
    while x<3
        x = x+1;
    println!("x={x}");
}
```

C/C++

```c
void main() {
    int x = 1;
    while (x<3)
        x+=1;
    printf("x = %d", x);
}
```

Error

```
error: expected `{`, found `x`
 --> src\main.rs:6:9
  |
5 |     while x<3
  |     ----- --- this `while` condition successfully parsed
  |     |
  |     while parsing the body of this `while` expression
6 |         x = x+1;
  |         ^ expected `{`
  |
help: try placing this code inside a block
  |
6 |         { x = x+1; }
  |         +          +
```

The C/C++ code will compile

while

Both break and continue keywords can be used in a while statement, with the same logic as the one from C/C++ (break or continue the loop).

Rust

```rust
fn main() {
    let mut x = 1;
    while x < 10 {
        if x % 3 == 0 {
            break;
        }
        x = x + 1;
    }
    println!("{x}");
}
```

Output
3

while

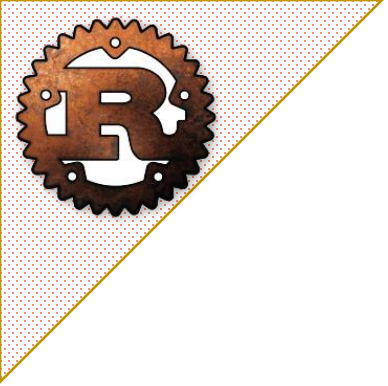
Let’s consider the following problem:

- Let there be a number of form $\overline{\mathbf{abc}}$, where a, b and c are digits between 1 and 9
- Can we find the smallest number of this form that has the following relation between a, b and c

    1) a = b x 2

    2) b = c x 2

The answer is simple → there are two numbers that respect this condition: 421 and 842, and as we are searching for the smallest one the final answer will be 421.



while

Let’s see how the previous problem can be solved in Rust:

Rust

```rust
fn main() {
    let (mut x, mut y, mut z) = (1, 1, 1);
    while x < 10 {
        y = 1;
        while y < 10 {
            z = 1;
            while z < 10 {
                if (x == y * 2) && (y == z * 2) {
                    println!("{x}, {y}, {z}");
                    break;
                }
                z += 1;
            }
            y += 1;
        }
        x += 1;
    }
}
```

This code will run correctly, but it will not print the smallest solutions but instead it will print **all solutions**.

Output

```
4, 2, 1
8, 4, 2
```

while

Let’s see how the previous problem can be solved in Rust:

Rust

```rust
fn main() {
    let (mut x, mut y, mut z) = (1, 1, 1);
    let mut done = false;
    while (x < 10) && (!done) {
        y = 1;
        while (y < 10) && (!done) {
            z = 1;
            while (z < 10) && (!done) {
                if (x == y * 2) && (y == z * 2) {
                    println!("{x}, {y}, {z}");
                    done = true;
                }
                z += 1;
            }
            y += 1;
        }
        x += 1;
    }
}
```

One solution will be to **create a flag** variable that forces the exit from every inner while loops. Once the first solution is found, we enable that flag and for the exit from every inner while loop.

Output

```
4, 2, 1
```

while

Rust has a way of providing a name (a label) for every loop statement (for, while or loop).

This is done via the following format:

- `'<name>: <for|while|loop>`

Example: `'first_while: while …`

This allows keywords like break or continue to explicitly say if we want to break the current loop or if we want to break a specific loop based on the loop name / label.

  - break
  - break `'first_while`

while

Let’s see how the previous problem can be solved in Rust:

Rust

```rust
fn main()
{
    let (mut x, mut y, mut z) = (1, 1, 1);
    'first_while: while x < 10 {
        y = 1;
        'second_while: while y < 10 {
            z = 1;
            while z < 10 {
                if (x == y * 2) && (y == z * 2) {
                    println!("{x}, {y}, {z}");
                    break 'first_while;
                }
                z += 1;
            }
            y += 1;
        }
        x += 1;
    }
}
```

This is a more elegant solution as we can specify what while should the break keyword break;

When break ‘first_while is called, it will break the most outer while and stop the entire process.

Output

```
4, 2, 1
```

loop

Rust also has a special loop called loop statement

- *loop { … do block … }*

In a nutshell a loop statement is nothing but a while true {…} statement.

Rust

```rust
fn main() {
    let mut x = 0;
    loop {
        println!("{x}");
        if x >= 3 {
            break;
        }
        x += 1;
    }
}
```

C/C++

```c
void main() {
    int x = 1;
    while (true) {
        printf("x = %d", x);
        if (x>=3) break;
        x+=1;
    }
}
```

Output

```
0
1
2
3
```

loop

The main advantage of the loop statement stays in the fact that it can be transformed in an execution statement where the return of the loop can be obtained via a break <value> statement.

Format: let <variable>:<type> = loop { … break <value> … };

Rust

```rust
fn main() {
    let (mut x, mut y) = (24, 18);
    let cmmdc: i32 = loop {
        if x > y { x -= y; }
        else if y < x { y -= x }
        else { break x; }
    };
    println!("{cmmdc}");
}
```

Output
6

loop

Type can be omitted and will be inferred from the value returned via break statement.

Format: let <variable> = loop { … break <value> … };

Rust

```rust
fn main() {
    let mut sum = 0;
    let mut counter = 0;
    let first_10_sum = loop {
        if counter > 10 { break sum; }
        sum += counter;
        counter += 1;
    };
    println!("{first_10_sum}");
}
```

Output
55

If let

"if let" tries to match an expression with a specified pattern. If the expression matches the pattern, the assignment is being performed and the code from the <then block> is being executed. Otherwise, the <else> block, if present is executed.

- *if let <pattern> = <expression> { … then block … }*

or

- *if let <pattern> = <expression> { … then block … } else { … else block … }*

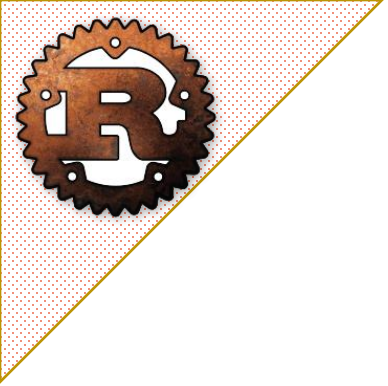
This statement is **NOT** to be confused with the if var=<expression> statement from C/C++, as they serve a different scope.

if let

The assignment (in case of *if let <pattern> = <expression> { …}* usually translates in a match of:

- An enum variant
- A structure with parameters
- Numerical constants
- Tuples
- …

We will discuss more about this type of statement when we talk about enums, errors and variants (as this is where this statement is mostly used).



if let

If let statement (example):

Rust

```rust
fn main() {
    let x = 10;
    if let 5 = x { println!("x is 5 "); }
    if let 10 = x { println!("x is 10"); }
    let tuple = (1,2); // a tuple with two integers 1 and 2
    if let (x,2) = tuple { println!("A tuple of {x} and 2");}
    if let (x,1) = tuple {
        /* do something */
    } else {
        println!("current tuple is not in the form (x,1)");
    }
}
```

Output

```
x is 10
A tuple of 1 and 2
current tuple is not in the form (x,1)
```

if let

If let statement (example):

Rust

```
fn main() {
    let x = 10;
    if let 5 = x { println!("x is 5 "); }
    if let 10 = x { println!("x is 10"); }
    let tuple = (1,2); // a tuple with two integers 1 and 2
    if let (x,2) = tuple { println!("A tuple of {x} and 2");}
    if let (x,1) = tuple {
        /* something */
    } else {
        println!("tuple is not in the form (x,1)");
    }
```

This actually translates into the following logic:
* If variable **tuple's** second value is 2 then copy its first value into variable "x" and run the code from ***<then>*** block.

if let

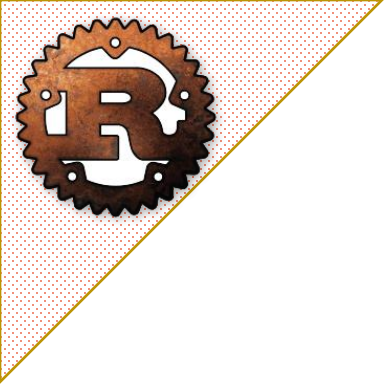
If let statement (example):

Rust

```
fn main() {
    let x = 10;
    if let 5 = x { println!("x is 5 "); }
    if let 10 = x { println!("x is 10"); }
    let tuple = (1,2); // a tuple with two integers 1 and 2
    if let (x,2) = tuple { println!("A tuple of {x} and 2");}
    if let (x,1) = tuple {
        /* do something */
    } els
        p
    }
}
```

An equivalent code will look like this:

```
// if the second value from a tuple is 2
if tuple.1 == 2 {
    // assign the first value from the tuple to variable x
    let x = tuple.0;
    println!("A tuple of {x} and 2");
}
```



if let

Let’s discuss another example:

Rust

```
fn main() {
    if let v = 0 {
        println!("{v}");
    }
}
```

Output

```
0
```

Warning

```
warning: irrefutable `if let` pattern
 --> src\main.rs:2:8
  |
2 |     if let v = 0 {
  |        ^^^^^^^^^
  |
  = note: this pattern will always match, so the `if let` is useless
  = help: consider replacing the `if let` with a `let`
  = note: `#[warn(irrefutable_let_patterns)]` on by default
```

In this example, “if let v = 0” will always be true (in reality there is no pattern to match here – just a simple assignment). The code will compile, but this no different than just writing “let v = 0”.

if let

Let’s discuss another example:

Rust

```rust
fn main() {
    let v = 20;
    if let v = 0 {
        println!("{v}");
    }
    println!("{v}");
}
```

Output

```
0
20
```

This is a similar logic → however, notice that the “v” variable from the if let statement has a ***limited lifetime*** to the *<then> block* and will not affect the outer “v” variable. As a result, the second println will print the value 20.

if let

Let’s discuss another example:

Rust

```rust
fn main() {
    let v = 20;
    if let v = 0 {
        println!("inner = {v}");
    }
    println!("outer = {v}");
}
```

Output

```
Inner = 0
outer = 20
```

C++

```cpp
#include <stdio.h>
int main() {
    int v = 20;
    if (v = 0) {
        printf("inner = %d",v);
    }
    printf("outer = %d",v);
}
```

Output

```
outer = 0
```

Keep in mind that these two code are **NOT EQUIVALENT**. In case of C++ example, variable v is first instantiated with value 0 and then evaluated (and since 0 = false) the ***<then block>*** is not executed. Furthermore, it’s the same variable “v” and as such the second printf will print value 0.

while let

The while let form can be used in a loop with a similar logic (pattern must match in order for the loop to run).

Rust

```rust
fn main() {
    while let mut y = 0 {
        if y >= 3 { break; }
        y = y + 1;
        println!("{y}");
    }
}
```

On the first glance, we would expect this while to run for 3 iterations , print values from 1 to 3 and exit.

This is not to be confused with while var=expression statement from C/C++.

while let

The while let form can be used in a loop with a similar logic (pattern must match in order for the loop to run).

Rust

```rust
fn main() {
    while let mut y = 0 {
        if y >= 3 { break; }
        y = y + 1;
        println!("{y}");
    }
}
```

On the first glance, we would expect this while to run for 3 iterations , print values from 1 to 3 and exit.

This is not to be confused wit

from C/C++.

In reality, the code runs indefinitely. The initialization of y to 0 is evaluated on every iteration and as a result y will always be 0 and as such the break condition will not be achieved.

while let

The while let form can be used in a loop with a similar logic (pattern must match in order for the loop to run).

Rust

```
fn main() {
    while let mut y = 0 {
        if y >= 3 { break; }
        y = y + 1;
        println!("{y}");
    }
}
```

Warning

```
warning: irrefutable `while let` pattern
 --> src\main.rs:4:11
  |
4 |     while let mut y = 0 {
  |           ^^^^^^^^^^^^
  |
  = note: `#[warn(irrefutable_let_patterns)]` on by default
  = note: this pattern will always match, so the loop will never exit
  = help: consider instead using a `loop { ... }` with a `let` inside it
```

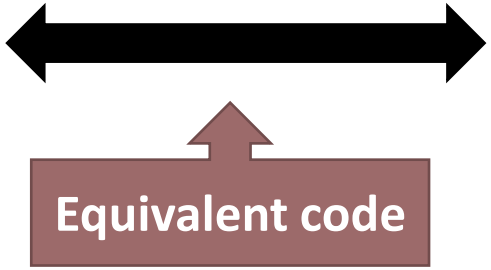
Rust notifies about this behavior through a warning !

while let

The while let form can be used in a loop with a similar logic (pattern must match in order for the loop to run).

Rust

```rust
fn main() {
    while let mut y = 0 {
        if y >= 3 { break; }
        y = y + 1;
        println!("{y}");
    }
}
```



Rust

```rust
fn main() {
    while true {
        let mut y = 0;
        if y >= 3 { break;}
        y = y + 1;
        println!("{y}");
    }
}
```

This is not to be confused with while var=expression statement from C/C++.

while let

The assignment (in case of *while let <pattern> = <expression> { …}* usually translates in a match of:

- An enum variant
- A structure with parameters
- Numerical constants
- Tuples
- …

We will discuss more about this type of statement when we talk about enums, errors and variants (as this is where this statement is mostly used).

let … else

“let … else” tries to match an expression with a specified pattern. If the expression matches the pattern, the assignment is being executed. Otherwise, an error (that will be discuss in the next courses) will be thrown:

let <pattern> = <expression> else { … error … }

This is mostly used with enum , variants or structs and we will further discuss this type of behavior at that point.

let … else

Some examples of “let … else”:

Rust

```rust
fn main() {
    let tuple = (1,2);
    let (x,2) = tuple else {
        panic!("Fail to assign !")
    };
    println!("{x}");
}
```

Output

```
1
```

Rust

```rust
fn main() {
    let tuple = (1,3);
    let (x,2) = tuple else {
        panic!("Fail to assign !")
    };
    println!("{x}");
}
```

Error

```
thread 'main' panicked at 'Fail to assign !',
src\main.rs:3:31
stack backtrace:
   0: std::panicking::begin_panic_handler
```

let … else

Some examples of “let … else”:

Rust

```
fn main() {
    let tuple = (1,2);
    let (x,2) = tuple else {
        panic!("Fail to assign !")
    };
    println!("{x}");
}
```

This code compiles and runs without any error. Since tuple variable matches the format **(<number>,2)**, the value of the first field from the tuple will be copied to variable “x”.

Rust

```
fn main() {
    let tuple = (1,3);
    let (x,2) = tuple else {
        panic!("Fail to assign !")
    };
    println!("{x}");
}
```

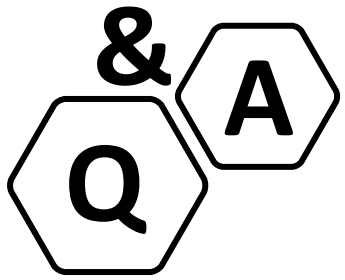
Notice that tuple variable is (1,3) and does not match the let … else requirement and as such a runtime error (panic) is thrown.

other statements

There are other more complex statement in Rust, such as:

- **for** (equivalent for classical for from C/C++ and a foreach)
- **match** (an equivalent for ***switch*** in C/C++ but more oriented to pattern matching)

As all these statements are either more complex or require understanding of different concepts in Rust, we will discuss them during the next courses.


&
Q
A