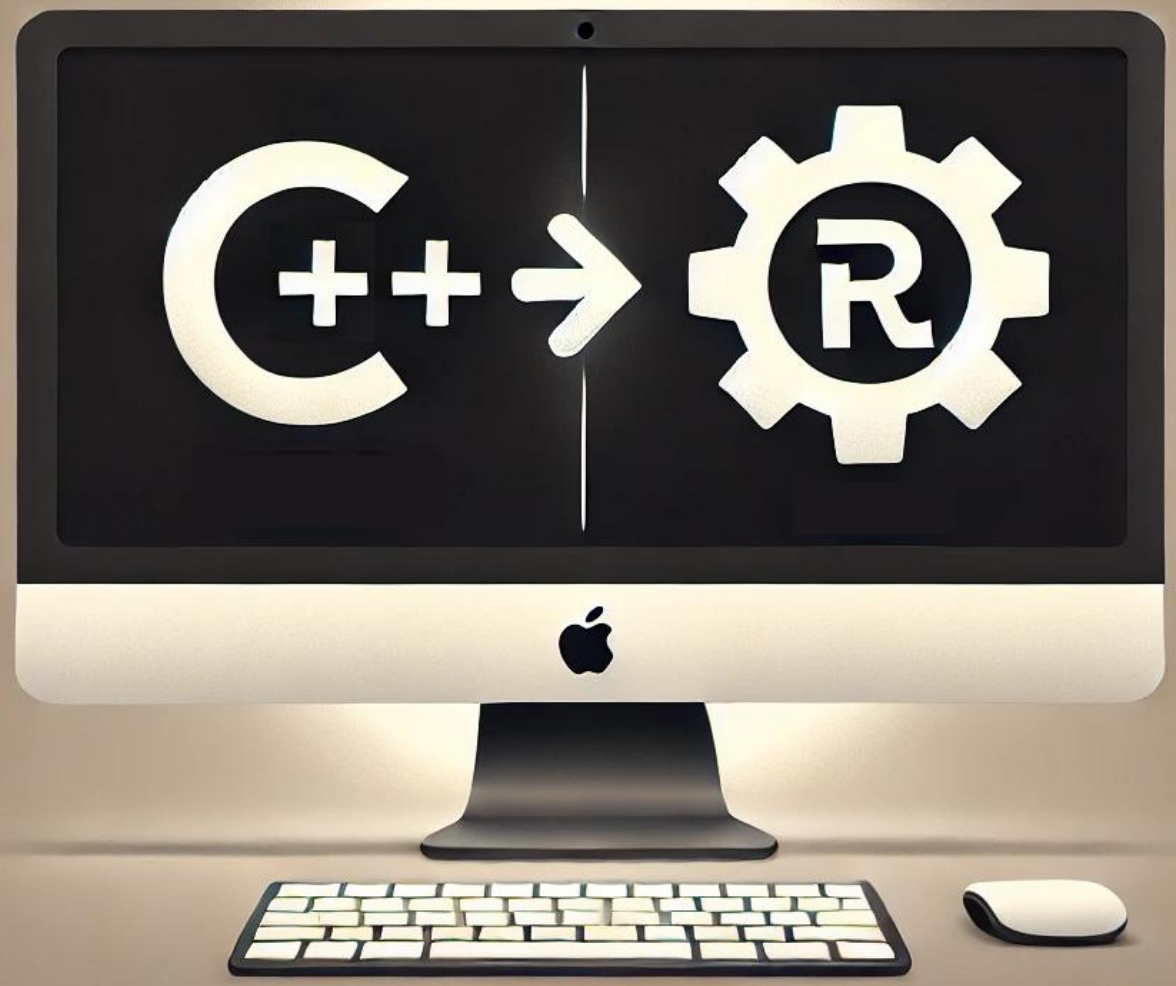
From C++ to Rust



Gavrilut Dragos

RUST

Philosophy

"You fight with the compiler, so you don’t fight with runtime bugs."

A little bit of history …

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:

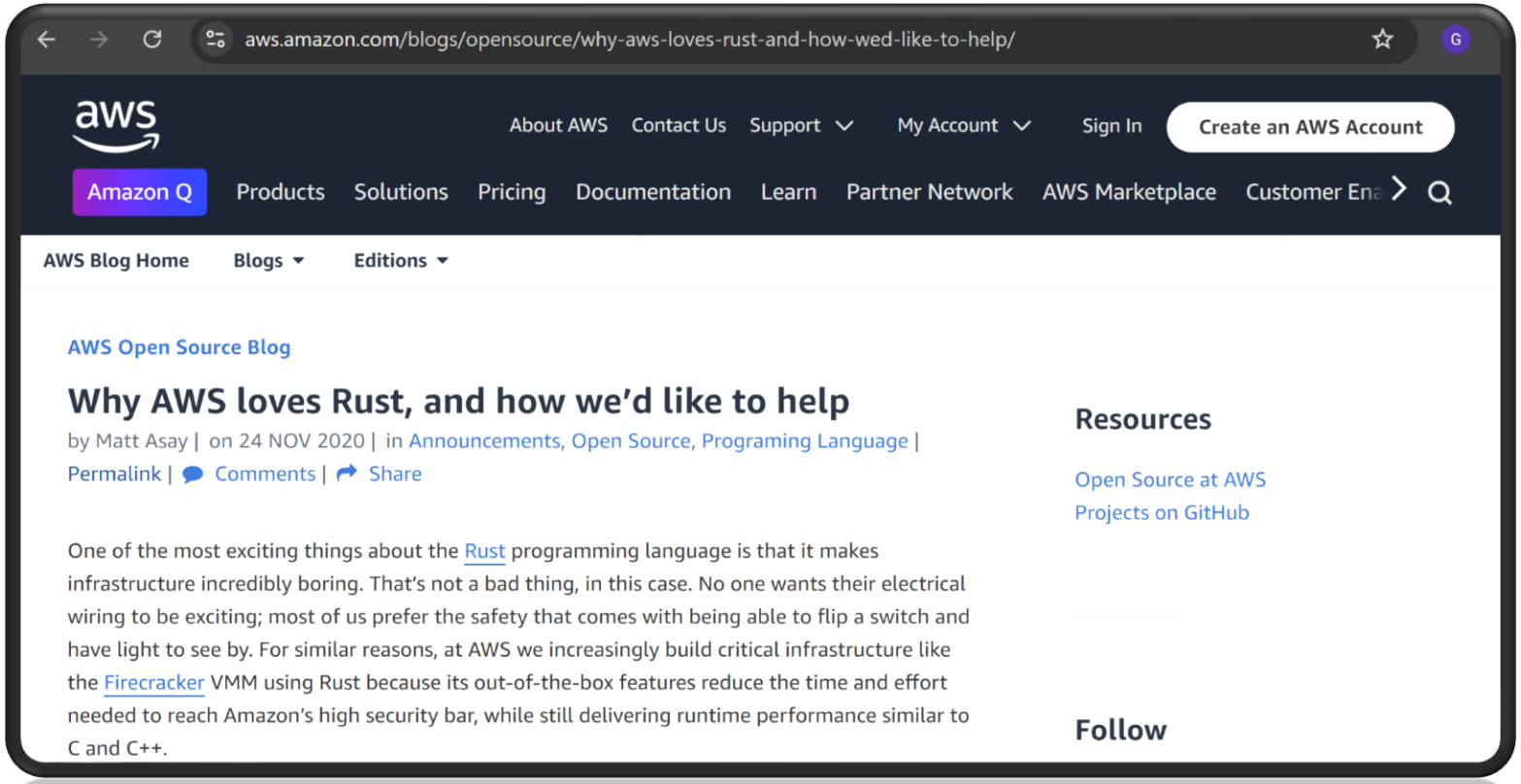
- Linux & Mac/OSX: run `curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh`
- GitHub repo: https://github.com/rust-lang/rust
- Windows install link: https://www.rust-lang.org/tools/install
- Documentation: https://doc.rust-lang.org/book/
- Quick install: https://rustup.rs/
- Official site: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

Rust 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- 2021 → Rust Foundation is formed, and the project is no longer maintained solely by Mozilla.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

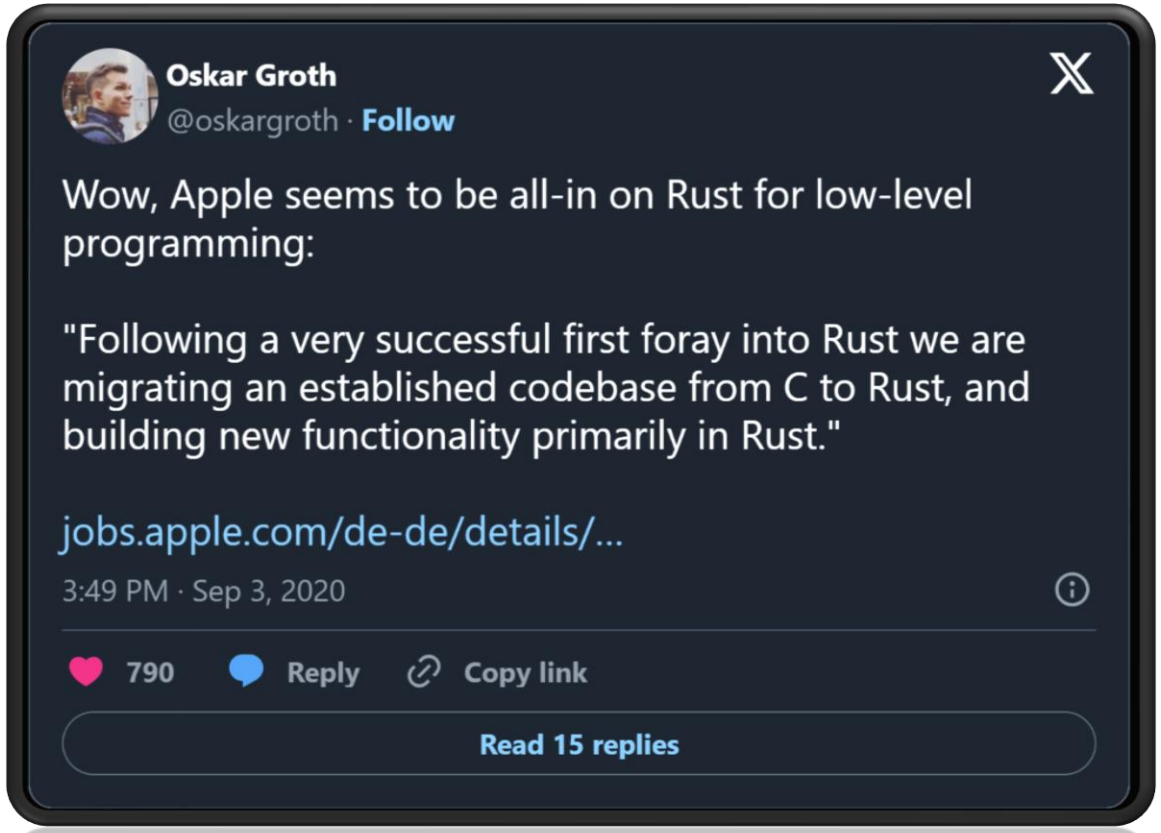
Rust History

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



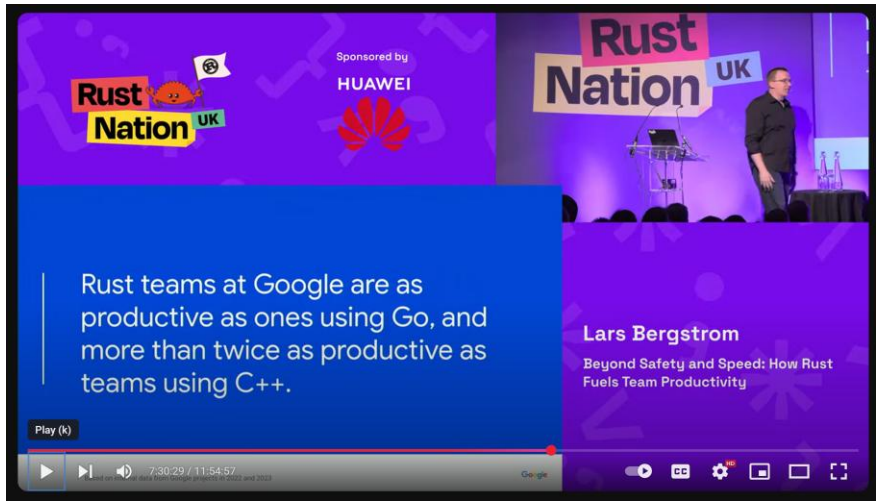
Rust History

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



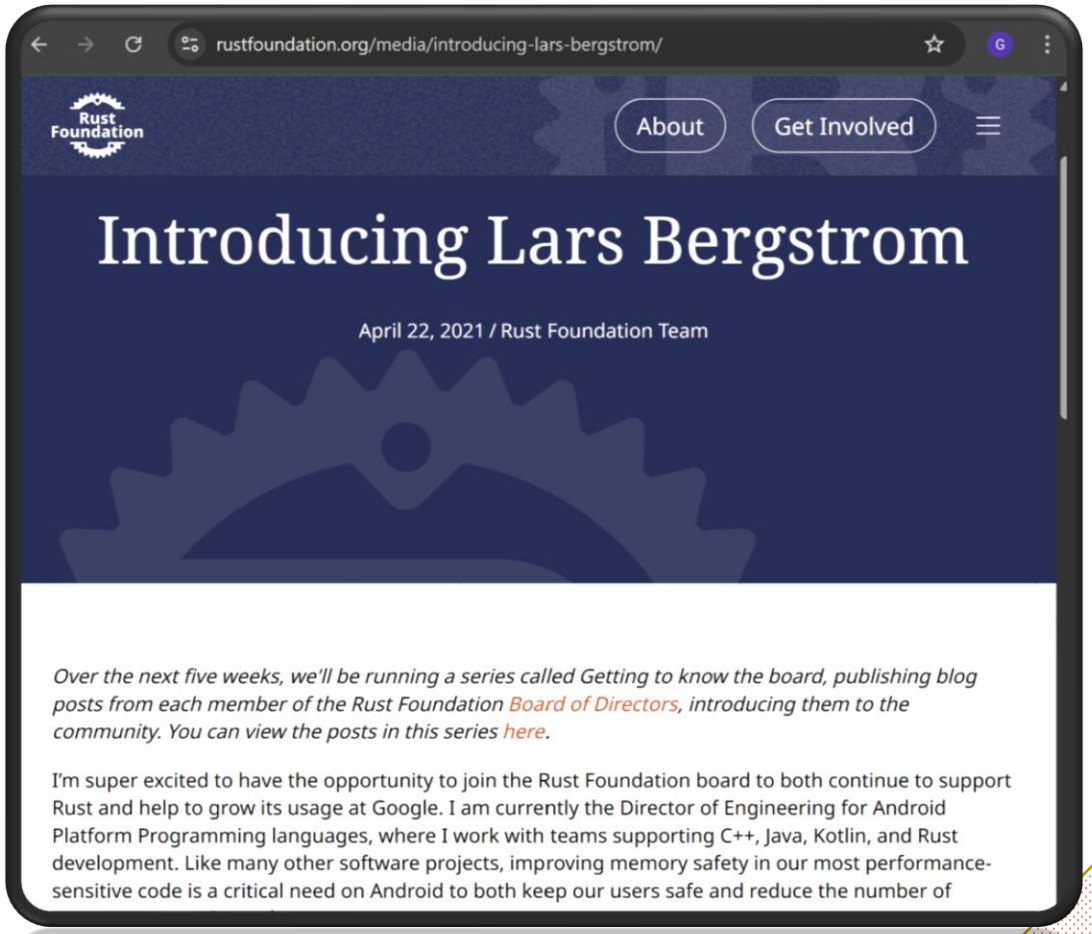
Rust History

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



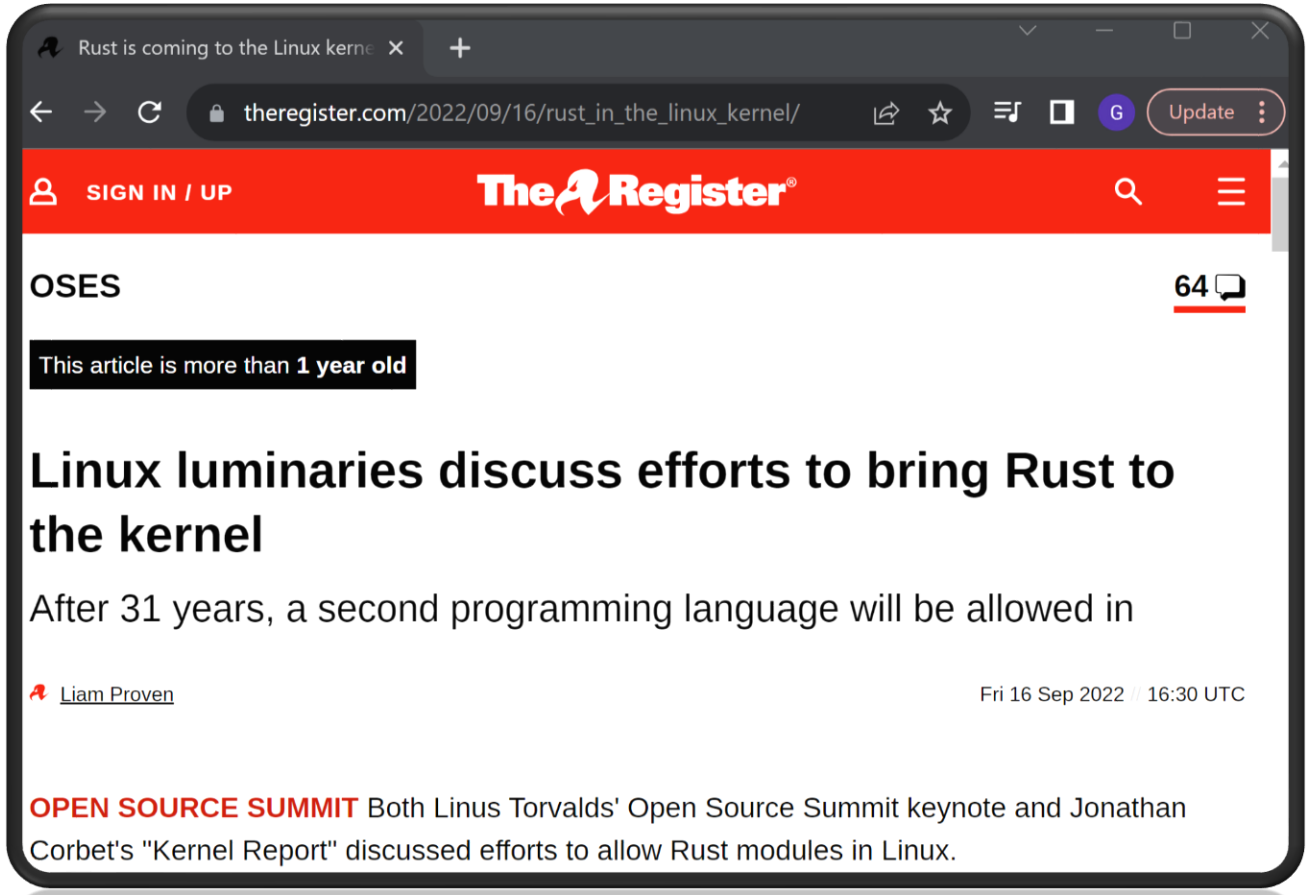
Rust nation UK (2024)

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



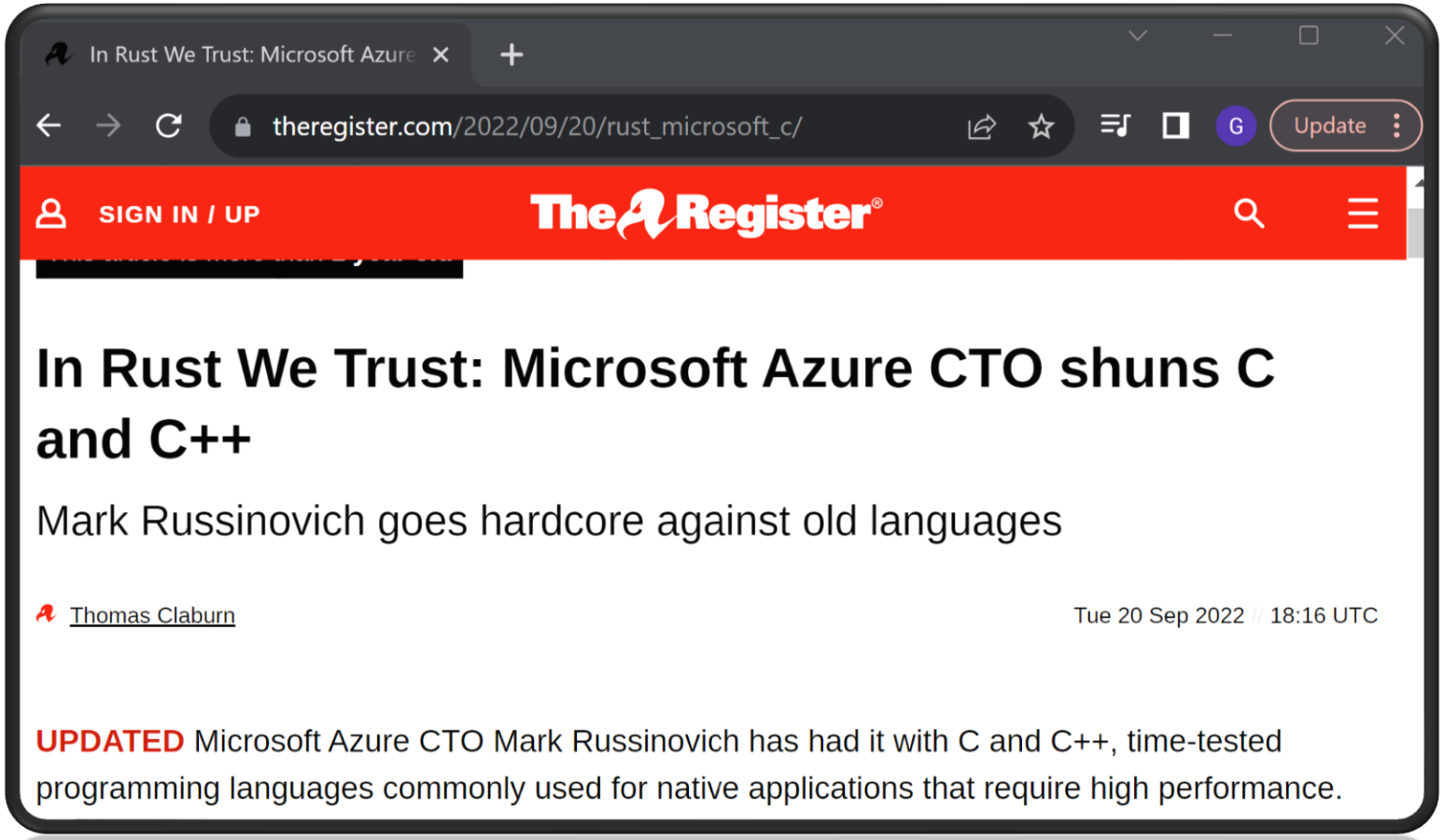
Rust History

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



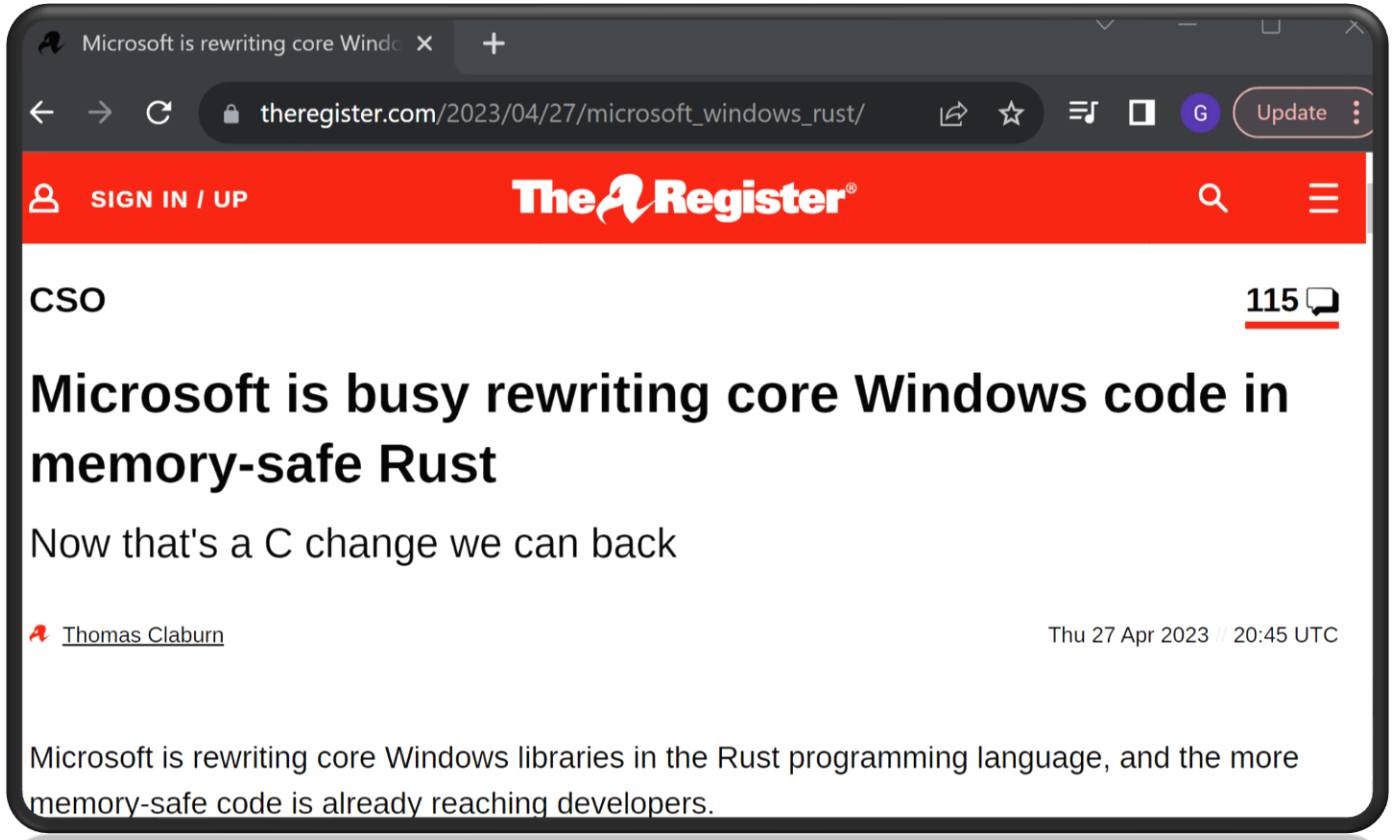
Rust History

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



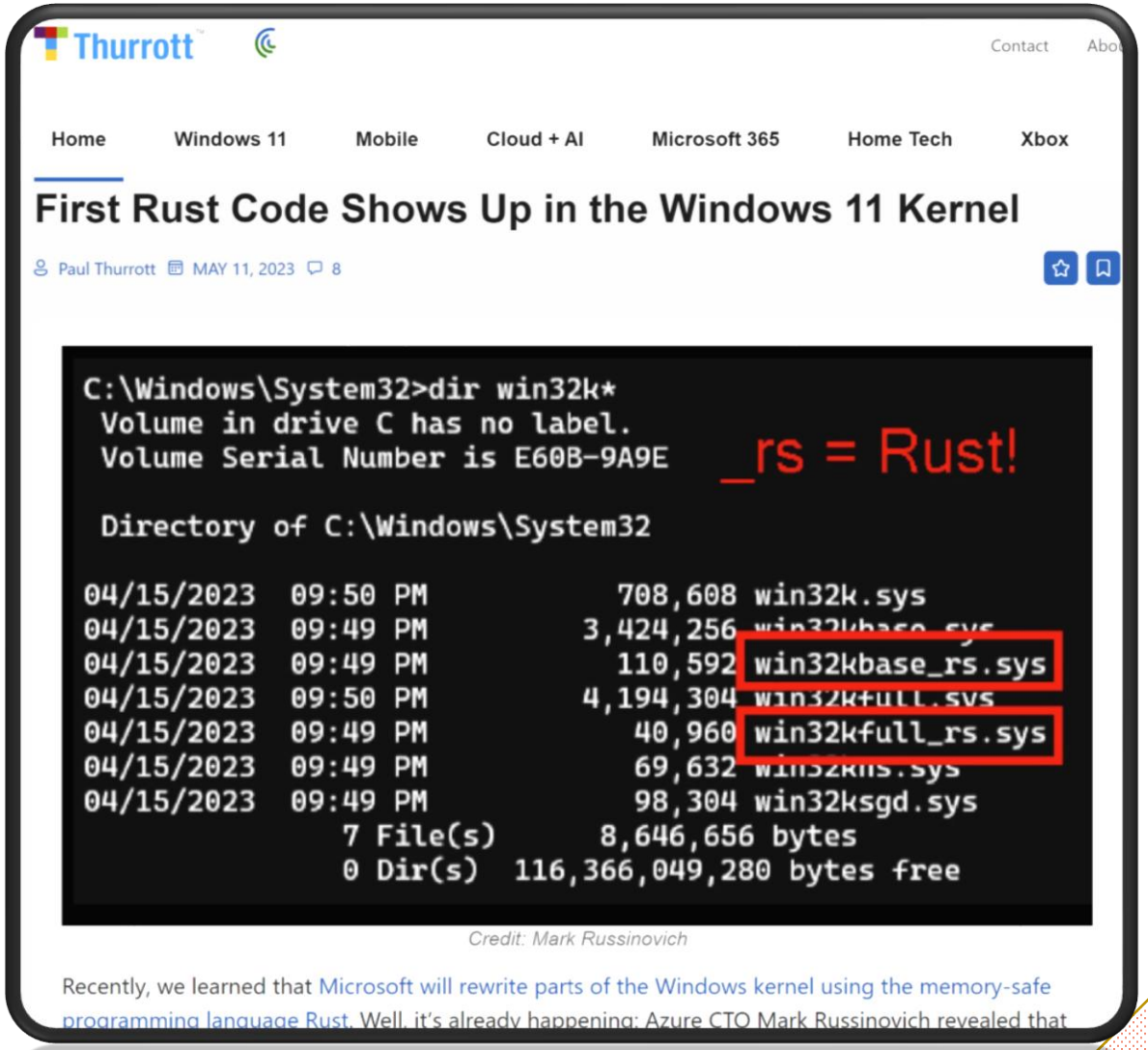
Rust History

- 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

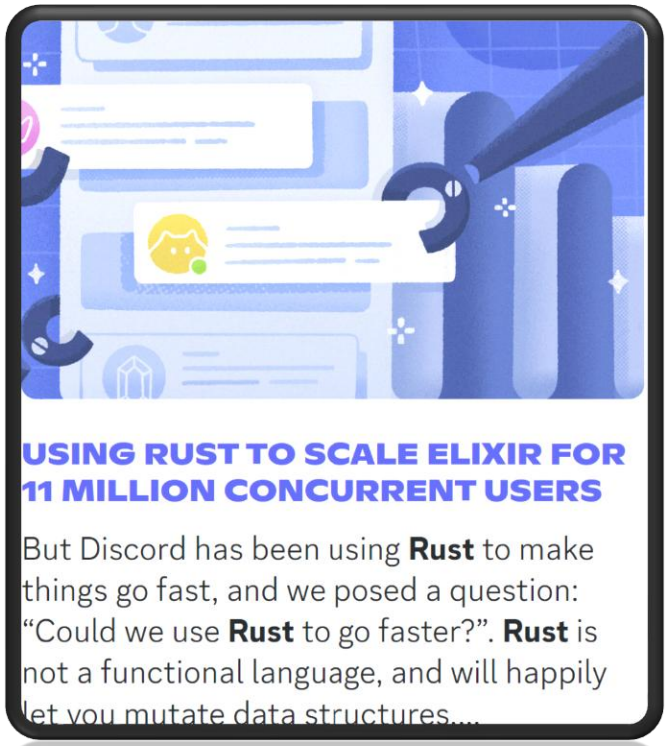
- In May.2024, Microsoft donates 1M USD to Rust Foundation to confirm company interest in this language.

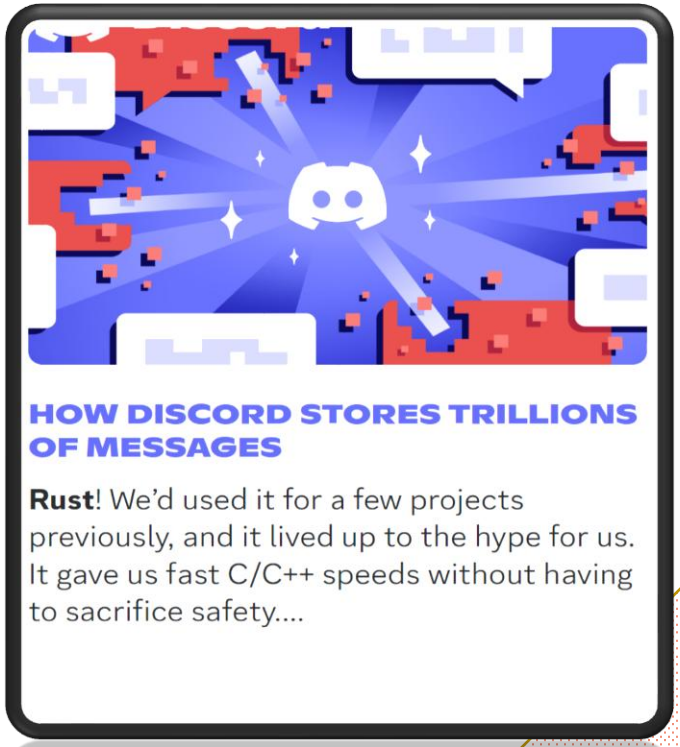
Rust History

- 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





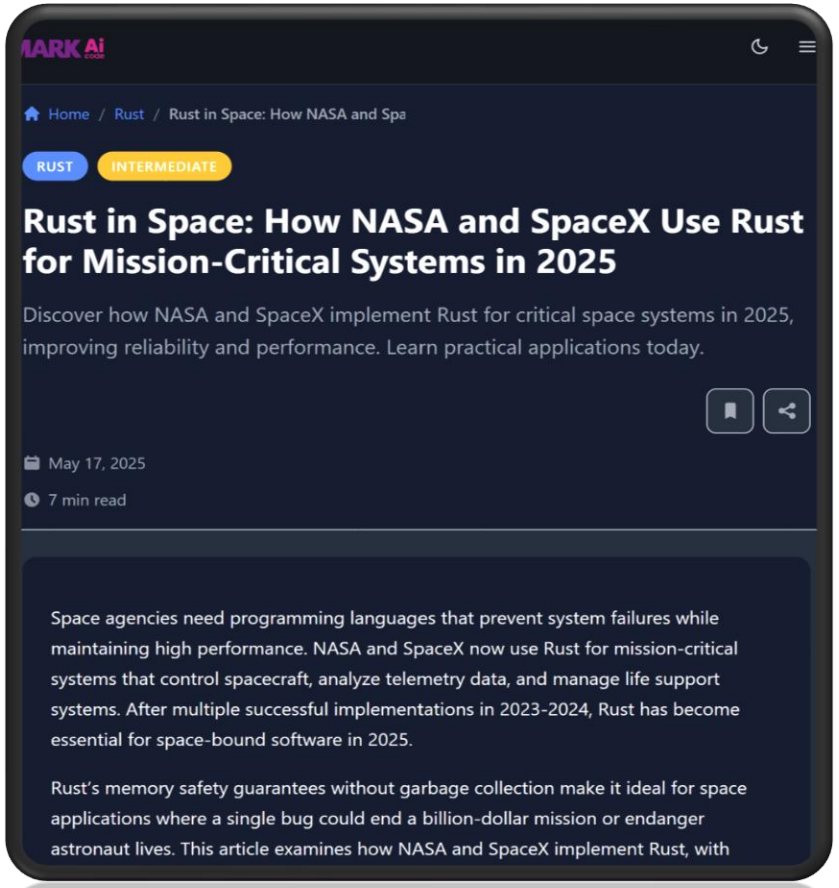
Rust History

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

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

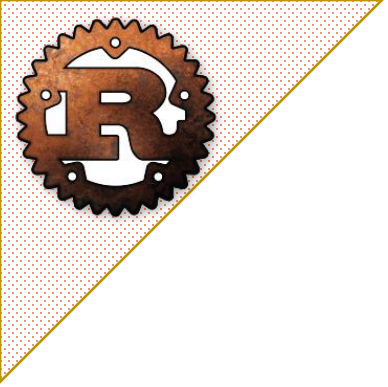
Create your very first RUST program

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:

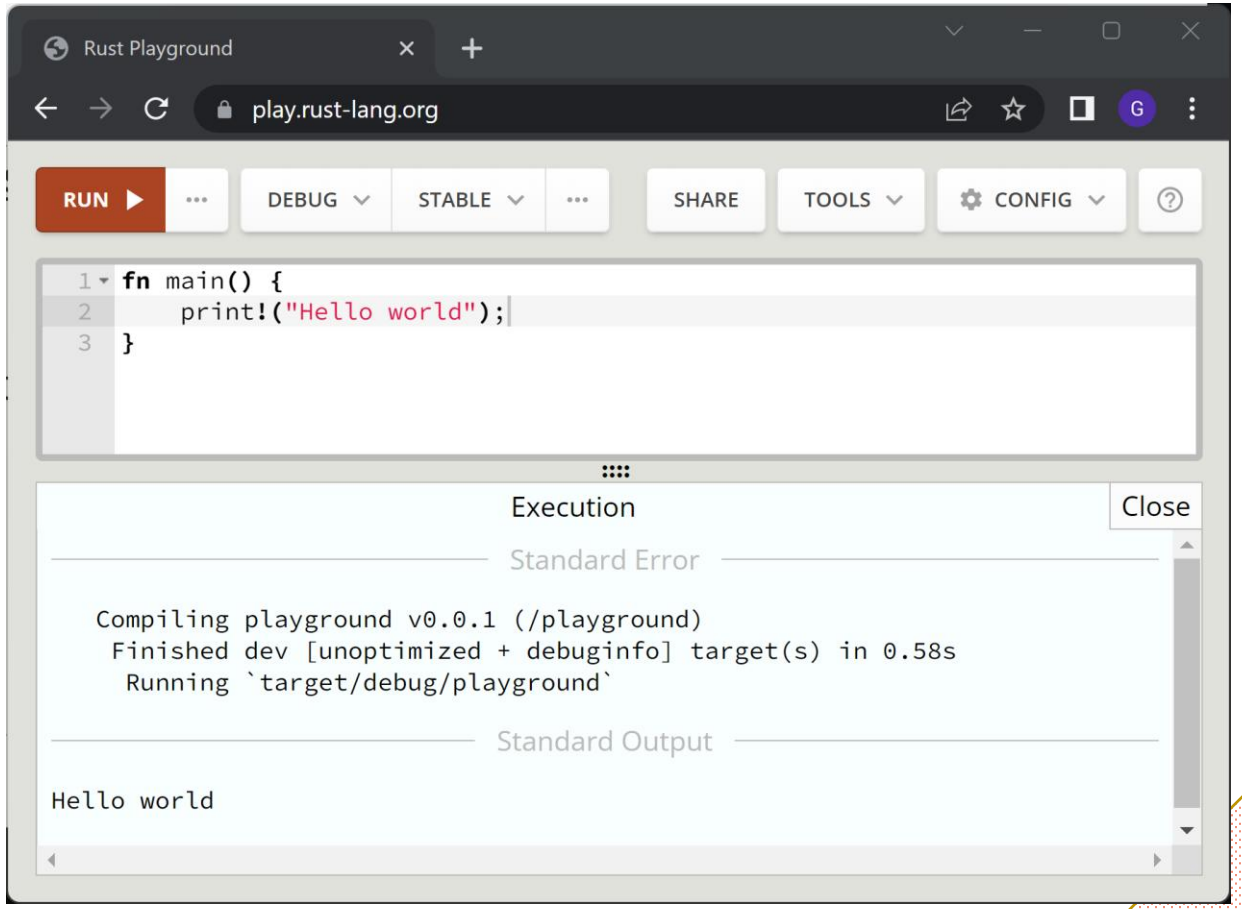
- Make sure that rust is installed
- Run the following command from command line: cargo new first
- You should see a new folder (named first) that was created in the current folder.
- Modify the "**\src\main.rs**" to contain the hello world example
- In folder "**\first**" execute the following command: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

Rust vs other languages

Learning Curve

Language	Learning Curve	Key Challenges	Ideal For
Python	Easy	Minimal; focuses on readability and simplicity	Beginners, data science, web development
JavaScript	Moderate	Asynchronous programming, dynamic typing	Web development, full-stack applications
Go	Moderate	Understanding goroutines and channels for concurrency	Backend services, cloud applications
Java	Moderate	Verbose syntax, object-oriented concepts	Enterprise applications, Android development
C++	Difficult	Manual memory management, complex syntax	Systems programming, game development
Rust	Very Difficult	Ownership model, lifetimes, borrow checker	Systems programming, performance-critical apps

Learning Difficulty

Language	Difficulty
HTML / CSS	10 / 100
Python	20 / 100
Ruby	30 / 100
JavaScript	40 / 100
C#	45 / 100
Switft	50 / 100
PHP	55 / 100
Go	60 / 100
Kotlin	65 / 100
Java	70 / 100
C++	75 / 100
Rust	80 / 100

https://www.crossover.com/resources/12-programming-languages-ranked-by-difficulty-chart

Rust vs C++

Rust vs C++

UN-INITIALIZED VARIABLES: In rust all variables, data members from structure, enum variants, etc MUST be initialized (if they are not the code will not compile)

Rust

```rust
fn main() {
    let a: i32;
    println!("{a}");
}
```

Won't compile → "a" was not initialized

C/C++

```cpp
int main() {
    int a;
    printf("%d", a);
}
```

Compiles OK ! (behavior can be changed with /SDl flag)

Rust vs C++

MISSING RETURN TYPE: In rust all functions / methods must return a value if it is being specified in the method / function definition. Failing to do this results in a compiler error.

Rust

```rust
fn foo(x: i32, y: i32) {

}

fn main() {
    let a: i32 = foo(10,20);
}
```

Won't compile → "foo" must return a i32

C/C++

```cpp
int foo(int x, int y) {

}
int main() {
    int a = foo(10, 20);
}
```

Compiles OK ! (release mode)

Rust vs C++

IMPLICIT CASTS: In rust you CAN NOT convert a number of a different type to another type ***IMPLICITELLY***. You need to explicetely specify the conversion.

Rust

```rust
fn main() {
    let a: bool = 123.5;
}
```

Won't compile → 123.5 is not a bool value

C/C++

```cpp
int main() {
    bool x = 123.10;
}
```

Compiles OK ! (with warning)

Rust vs C++

EXHAUSTIVENESS CHECKING: In Rust, in a ***match*** *(switch in C++)* ***block*** all possible values must be addressed. Failing to do so results in a compiler error.

Rust

```rust
fn main() {
    let x: u8 = rand();
    match x {
        0 => print!("0"),
        1 => print!("1"),
        2 => print!("2"),
        3 => print!("3"),
    };
}
```

Won't compile → values from 4 to 255 were not addressed.

C/C++

```c
int main() {
    unsigned char x = rand();
    switch (x) {
        case 0: printf("0"); break;
        case 1: printf("1"); break;
        case 2: printf("2"); break;
        case 3: printf("3"); break;
    }
}
```

Compiles OK !

Rust vs C++ (Variables & Functions)

Feature	C++	Rust
Variable implicit mutable state	Mutable	Immutable
Const type	Means a constant numerical value or a value that can not be changed	Means a constant numerical.
Overflow operations	**Allowed, not managed**	**Managed on DEBUG**
Uninitialized variables	**Yes (possible)**	**Not Possible**
Missing return type for functions	**Yes (possible)**	**Not Possible**
Basic types with clear size	**Partial** (e.g. int, char might have different representations)	**Yes (i8, i16, i32, … u32, u64, …)**
Larger types (128 bytes)	**No**	**Yes (i128 and u128)**
Implicit casts between basic types	**Yes**	**No (not allowed)**
String support	**Multiple (Ascii, WTF-16/32)**	**UTF-8 (guaranteed)**
Exhaustiveness Checking	**Not enforced**	**Enforced** at compile time

Rust vs C++

OWNERSHIP: A memory zone / a resource in Rust has ONLY ONE owner (or more simply put, you can not have two variables that share the same memory).

Rust

```rust
fn main() {
    let a = Box::new(5);
    let b = a;

    println!("{}", a);
}
```

Won't compile → "a" does not have ownership anymore.

C/C++

```cpp
int main() {
    int* a = new int(5);
    int* b = a;
    printf("%d", a);
}
```

Compiles OK !

Rust vs C++

MOVE SEMANTICS: Most of the types in Rust use move-semantics (an object is moved (ownership is transferred) and its lifetime ends (the source object is destroyed).

Rust

```rust
fn process(s: String) {
    println!("S = {s}");
}
fn main() {
    let s = String::from("ABC");
    process(s);
    println!("{s}");
}
```

Won't compile → println!("{s}") won't wok because "s" does not exists anymore

C/C++

```cpp
#include <string>
#include <iostream>

void process(std::string s) {
    std::cout << s;
}
int main() {
    std::string s = "ABC";
    process(s);
    std::cout << s;
}
```

Compiles OK !

Rust vs C++

BORROWING LOGIC: An object can borrow either multiple immutable references or **ONE** and only **ONE** mutable reference (every other combination is not allowed).

Rust

```rust
fn main() {
    let mut s = String::from("ABC");
    let r1 = &s;
    let r2 = &mut s;
    println!("{r1}");
}
```

Won't compile → we have both a mutable and immutable references at the same time

Rust

```rust
fn main() {
    let s = String::from("ABC");
    let r1 = &s;
    let r2 = &s;
    println!("{r1}");
}
```

Compiles OK !
(multiple immutable references)

Rust vs C++

LIFETIME LOGIC: References have lifetime associated with lifetime rules and relations that Rust uses the validate if a reference exists (is being used outside its lifetime / scope)

Rust

```rust
fn foo<'a>(a: &'a str,
           b: &'a str) -> &'a str {
    a
}
fn main() {
    let result;
    {
        let s1 = String::from("h");
        let s2 = String::from(“w");
        result = foo(&s1, &s2);
    }
    println!("{}", result);
}
```

Won’t compile → because result lives more than s1 and s2 (longer lifetime)

C/C++

```cpp
const char* foo(const char* a, const char* b)
{
    return a;
}
int main() {
    const char* result;
    {
        std::string s1 = "h";
        std::string s2 = “w";
        result = foo(s1.c_str(), s2.c_str());
    }
    std::cout << result;
}
```

Compiles OK !
(even if result points to an invalid memory address)

Rust vs C++ (Memory Overview)

Feature	C++	Rust
Manual Memory	new/delete or smart pointers (unique_ptr, etc.)	Ownership and borrowing system (no manual free)
Garbage Collector	**No (manual delete object)**	**No (Automatic clean up)**
Undefined Behavior	**Yes (possible)**	**Not Possible**
Use after free	**Yes (possible)**	**Not Possible**
NULL pointer associated risks	**Yes (possible)** (via NULL or nullptr)	**Not Possible** (there is no null pointer in Rust)
Dangling pointers	**Yes (possible)**	**Not Possible**
Lifetime support	**No**	**Yes (implicit for each reference)**
Thread safety	**Manual**	**Enforced at compile time**
Exception support	**Yes**	**No**
Error handling	**Not enforced**	**Enforced** at compile time via Result<T,R> or Option<T>

Rust vs C++

ALGEBRAIC DATA TYPE: An ***enum*** in Rust can have different data types for each of its fields. It can also be a regular enum (just like the ones from C/C++)

Rust

```rust
enum Color {
    Red,
    Green,
    Blue,
}

enum ComplexColor {
    Red,
    Green,
    Blue,
    RGB(u32),
    ARGB(u8,u8,u8,u8),
    Named(String)
}
```

C/C++

```cpp
enum Color {
    Red,
    Green,
    Blue,
}
```

Rust vs C++

SELF-CONSUME: In Rust you can create a method in a struct that can consume the instance (e.g. make sure that after the execution of that method the instance is destroyed).

Rust

```rust
struct Game {
    score: i32,
}
impl Game {
    fn run(self) {}
}

fn main() {
    let g = Game { score: 10 };
    g.run();
    g.run();
}
```

Won't compile → after the first call "g.run()", g variable does not exists any more

C/C++

```cpp
struct Game {
    int score;
    void Run() { }
};

int main() {
    Game g;
    g.score = 0;
    g.Run();
    g.Run();
    g.Run();
}
```

Compiles OK !
(even if the game should be run only once)

Rust vs C++ (Classes)

Feature	C++	Rust
Encapsulation	**private**, **protected**, **public** modifiers	via **pub** and **modules**
Extending existing class	**No**	**Yes**
Inheritance	**Yes**	**No**
Method overloading	**Yes (possible)**	**Not Possible**
Virtual methods / Polymorphism	**Yes (possible)**	**Yes (highly efficient)**
Interfaces	**Partial** (through abstract classes)	**Yes (with traits)**
Constructors	**Yes**	**No**
Copy/Move constructors/assign.	**Manual or Implcit (unsafe)**	**Yes (safe and fast)**
Destructors	**Yes**	**Yes (with traits – via drop)**
Friend functions	**Yes**	**No**
Operator overloading	**Yes**	**Yes**
Self consume capacity	**No**	**Yes**

Rust vs C++

TUPLES: Rust supports tuples (just like Python) that can be used to return any multiple data from any function. The same can be achived in C++ using std::pair (but it is not language specific)

Rust

```rust
fn foo(x: i32, y: i32) -> (i32, i32)
{
    (x + y, x * y)
}

fn main()
{
    let (s,p) = foo(10, 20);
}
```

C/C++

```cpp
#include <iostream>
#include <utility>

std::pair<int, int> foo(int x, int y)
{
    return {x + y, x * y};
}

int main() {
    auto [s, p] = foo(10, 20);
}
```

Rust vs C++

LAMBDA FUNCTIONS: In Rust you can create a method in a struct that can consume the instance (e.g. make sure that after the execution of that method the instance is destroyed).

Rust

```rust
fn main() {
    let mut x = 1;
    let print_x = || {
        println!("x={}",x);
        x+=1;
    };
    println!("x from main = {}",x);
    print_x();
}
```

Won't compile → because is is a mutable reference in a lambda (can not be used as a immutable reference in main)

C/C++

```cpp
int main() {
    int x = 1;

    auto print_x = [&x]() {
        std::cout << "x=" << x
                  << std::endl;
        x += 1;
    };
    std::cout << "x from main = "
              << x << std::endl;
    print_x();
    return 0;
}
```

Compiles OK !
(even if x is shared)

Rust vs C++

ITERATOR CHAIN: Rust allows the result of one iterator to be the input of another one. Using this techniques sequences of data can be easily converted / process.

Rust

```rust
fn main() {
    let a = vec![1,2,3,4,5,6,7,8,9];
    let s:i32 = a.iter()
                 .skip(3)
                 .take(4)
                 .inspect(|x| println!{"{:?}",*x})
                 .sum();
    println!("sum is {}",s);
}
```

C/C++

```cpp
int main() {
    std::vector<int> a = {1,2,3,4,5,6,7,8,9};
    auto begin = std::next(a.begin(), 3);
    auto end = std::next(begin, 4);
    int sum = 0;
    for (auto it = begin; it != end; ++it) {
        std::cout << *it << "\n";
        sum += *it;
    }
    std::cout << "sum is " << sum << "\n";
    return 0;
}
```

Rust vs C++ (Data structures)

Feature	C++	Rust
Tuples	**Yes (via std::pair)**	**Implicit (as part of the language)**
Type alias	**Yes (typedef & using)**	**Yes**
New type idiom	**Partial** (through wrapper classes)	**Yes**
Tree structures	**Yes**	**Yes**
Graph (maybe bi-oriented), Double linked list	**Yes**	**Complicated** **(due to ownership requirements)**
Vectors, Maps, Sets	**Yes (from std)**	**Yes (implicit - part of the Rust core)**
Iterator chaining	**Partial** (but supported)	**Yes**
Templates/Generics	**Yes** (With a limited constrains)	**Yes (with a complex constrain)**
Smart pointers	**Not implicit** (you need to use std)	**Implicit (as part of the language)**
Lambda	**Yes** (but unsafe)	**Yes (safe)**

Rust vs C++

MACROS: In Rust , macros are evaluated over the AST (Abstract Syntax Tree) and as such avoid several pitfalls that come with a simple subsitusion. Parameters must be explained (e.g. expression, const, etc)

Rust

```
macro_rules! multiply {
    ($p1: expr , $p2: expr) => {
        $p1 * $p2
    };
}
fn main() {
    let z = multiply!(1 + 2, 3 + 1);
    println!("{z}");
}
```

Compiles OK (and prints 12)

C/C++

```
#include <iostream>

#define MULTIPLY(x,y) x*y

void main() {
    auto z = MULTIPLY(1+2,3+1);
    printf("%d,%d,%d",x,y,z);
}
```

Compiles but prints 8
(8 = 1 + 2*3 + 1 = 1+6+1)

Rust vs C++

RECURSIVE MACROS: Rust macros support recursion. There are some limitations on how many such calls the compiler will do, but they can be overwritten by the use of `#![recursion_limit = "256"]` (with different values).

Rust

Compiles OK

```
macro_rules! count {
    () => {0usize};
    ($first:tt) => {1usize};
    ($first:tt,$($tail:tt),*)=> {
        1usize + count!($($tail),*)
    }
}
fn main() {
    let x = count!(1,2,3,4,5);
    println!("{x}");
}
```

Rust vs C++

PROCEDURAL MACROS: In Rust , you can write a form of compiler extension that creates Rust code that will further be compiled and executed

Rust

```rust
#[EnumBitFlags(bits=16]
pub enum Test_16bit {
    V1 = 1,
    V2 = 2,
    V3 = 4,
    V4 = 0x8000
}
```

Rust (resulted code)

```rust
#[derive(Copy, Clone, Debug)]
pub struct Test_16Bits { value: u16 }
impl Test_16Bits {
    pub const V1: Test_16Bits = Test_16Bits { value: 0x1u16 };
    pub const V2: Test_16Bits = Test_16Bits { value: 0x2u16 };
    pub const V3: Test_16Bits = Test_16Bits { value: 0x4u16 };
    pub const V4: Test_16Bits = Test_16Bits { value: 0x8000u16 };
    pub const None: Test_16Bits = Test_16Bits { value: 0 };

    pub fn contains(&self, obj: Test_16Bits) -> bool {…}
    pub fn contains_one(&self, obj: Test_16Bits) -> bool {…}
    pub fn is_empty(&self) -> bool {…}
    pub fn clear(&mut self) {…}
    pub fn remove(&mut self, obj: Test_16Bits) {…}
    pub fn set(&mut self, obj: Test_16Bits) {…}
    pub fn get_value(&self) -> u16 {…}
}
impl std::ops::BitOr for Test_16Bits {…}
impl std::ops::BitOrAssign for Test_16Bits {…}
impl std::ops::BitAnd for Test_16Bits {…}
impl std::ops::BitAndAssign for Test_16Bits {…}
impl std::cmp::PartialEq for Test_16Bits {…}
impl std::default::Default for Test_16Bits {…}
impl std::fmt::Display for Test_16Bits {…}
```

Rust vs C++ (Macros)

Feature	C++	Rust
Type	**Preprocessor (text substitution)**	**Hygienic (work at AST level)**
Compile-time execution	**No**	**Yes (expend into valid Rust code)**
Hygiene (name safety)	**No**	**Yes (prevents name collisions)**
Debug support	**No**	**Yes (compiler aware)**
Error messages	**Poor — errors reported after expansion**	**Excellent — errors often point to macro definition**
Parameter parsing	**Very weak — no parsing, just token substitution**	**Strong pattern matching and repetition with $() syntax**
Use cases	Constants, small functions, platform-specific code	DSLs, declarative code, boilerplate reduction, metaprogramming
Procedural macros (compiler extensions)	**No**	**Yes (powerful support)**
Recursive calls	**No**	**Yes (configurable recursive depth)**

Rust vs C++

DATA RACES: Rust can statically (at compile time) analyze cases where a reference is used in multiple threads (that could lead to data races) and not allow them.

Rust

```rust
use std::rc::Rc;
use std::thread;

fn main() {
    let data = Rc::new(42);

    let h = thread::spawn(move || {
        println!("Value = {}", data);
    });

    h.join().unwrap();
}
```

Won't compile → because data can not be moved from the main thread to another thread)

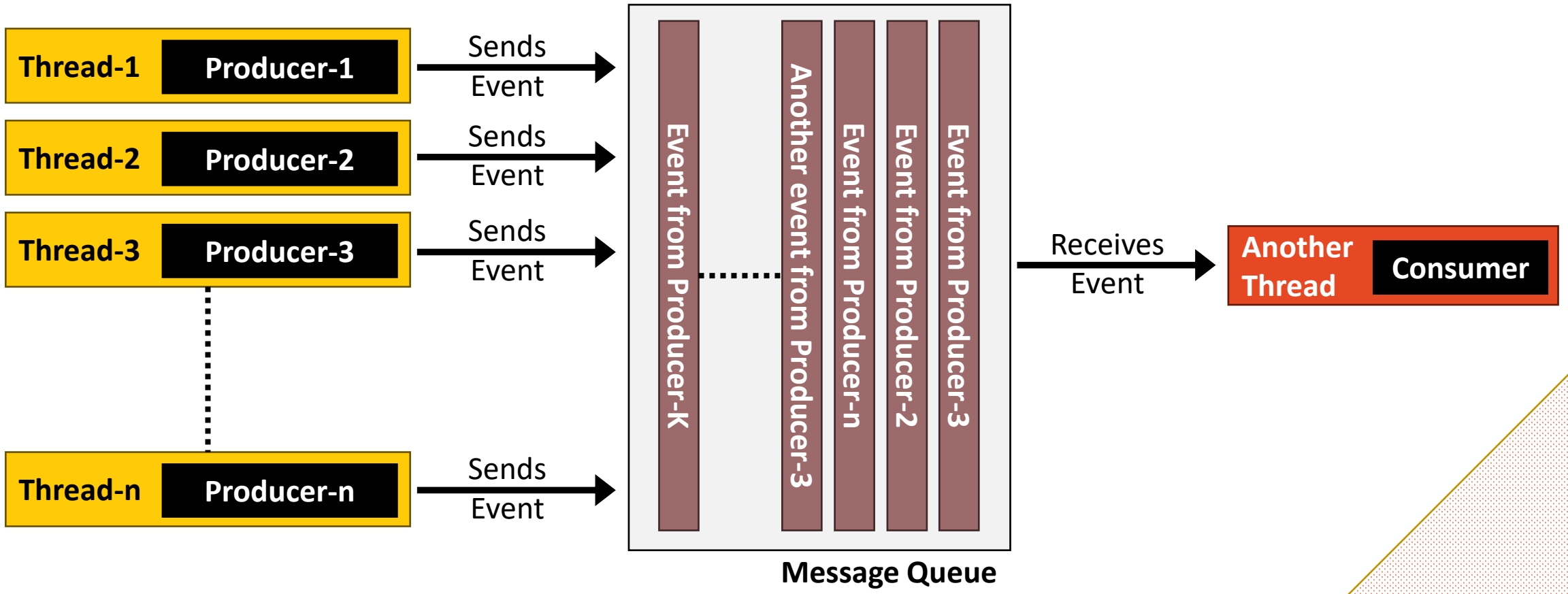
C/C++

```cpp
#include <thread>
int main() {
    int data = 42;

    std::thread handle([&data]() {
        std::cout << "Value = "
                  << data
                  << std::endl;
    });

    handle.join();
}
```

Compiles and Run !
(but allows access to a stack variable from a different thread)

Rust Channels

A channel is MPSC (**M**ultiple **P**roducers **S**ingle **C**onsumer) mechanism that can be used to send data from one thread to another.



Rust Channels

Using channels (example)

Rust (main)

```rust
use std::sync::mpsc;

struct Message { id: u32, text: String }
impl Message { fn new(id: u32, text: &str) -> Self {…} }

fn main() {
    let mut v = Vec::new();
    {
        let (producer, receiver) = mpsc::channel::<Message>();
        v.push(std::thread::spawn(move || loop {…}));
        for i in 0..5 {
            let th_producer = producer.clone();
            v.push(std::thread::spawn(move || {…}));
        }
    }
    for th in v { th.join().unwrap(); }
    println!("All threads finished");
}
```

Output (possible)

```
Received : Hello from 0
Received : Hello from 2
Received : Hello from 2
Received : Hello from 3
Received : Hello from 3
Received : Hello from 4
Received : Hello from 4
Received : Hello from 4
Received : Hello from 1
Terminate the receiver thread
All threads finished
```

Rust vs C++ (Synchronization)

Feature	C++	Rust
Data races	**Possible and undefined behavior**	**Not possible (safe)**
Mutex & Locks	**Yes**	**Yes**
Atomic types	**Yes**	**Yes**
Conditional variables	**Yes**	**Yes**
Channels	**No**	**Yes (part of the standard library)**
Async/await syntax	**No (maybe in C++26)**	**Yes**
Async runtimes	**No**	**Yes (tokio, async-std, etc)**
Green threads	**No**	**Yes - via async runtimes (non-blocking tasks, cooperative scheduling)**

Conditional Compilation

Let’s see a simple example (with code and cargo.toml).

cargo.toml

```
[package]
name = "my_app"
version = "0.1.0"
edition = "2021"

[features]
METHOD_A = []
METHOD_B = []
```

Rust

```
#[cfg(feature = "METHOD_A")]
fn foo() {
    println!("Method A");
}
#[cfg(feature = "METHOD_B")]
fn foo() {
    println!("Method B");
}
fn main() {
    foo();
}
```

C++ (equivalent code)

```
#ifdef METHOD_A
    void foo() {
        printf("Method A");
    }
#endif

#ifdef METHOD_B
    void foo() {
        printf("Method B");
    }
#endif

void main() {
    foo();
}
```

There are two options to run this program:

1. Run “cargo run --features METHOD_A”

Rust unit tests

Rust allows to write unit tests within your own codebase that validate that your code runs as expected

Rust (main.rs)

```rust
pub fn add(x: u8, y: u8) -> Option<u8> {
    let result = (x as u32) + (y as u32);
    if result > 255 {
        return None;
    }
    return Some(result as u8);
}
```

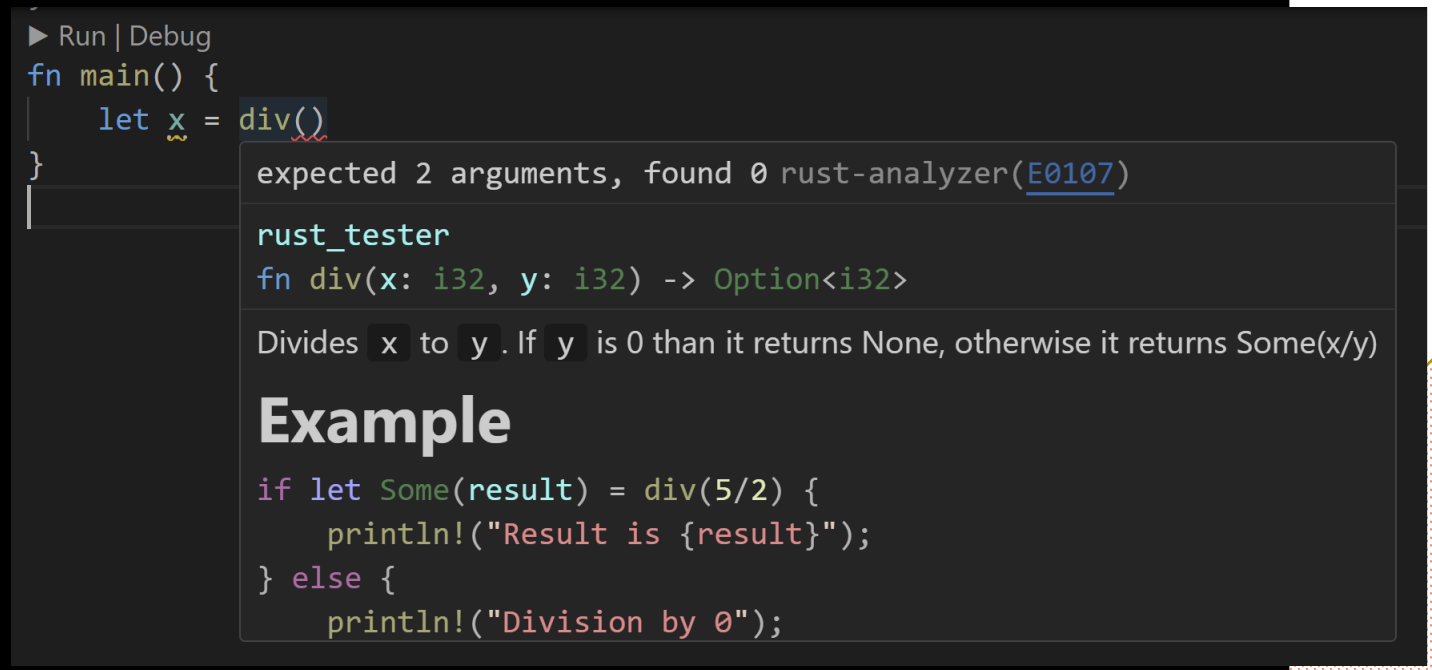
```rust
#[test]
fn check_add() {
    assert!(add(1,2)==Some(3));
    assert_eq!(add(100,155),Some(255));
    assert_ne!(add(0,0),Some(1));
}
#[test]
fn check_overflow() {
    assert_eq!(add(200,200),None);
    assert_ne!(add(100,100),None);
}
```

Rust Documentation (Function/Module)

To write a documentation, use /// characters (on multiple lines) to explain (in Markdown format what that function / module is doing).

Rust

```rust
/// Divides `x` to `y`. If `y` is 0 than it returns None,
/// otherwise it returns Some(x/y)
///
/// # Example
///
/// ```
/// if let Some(result) = div(5/2) {
///     println!("Result is {result}");
/// } else {
///     println!("Division by 0");
/// }
fn div(x: i32, y: i32) -> Option<i32> {
    if y != 0 {
        Some(x / y)
    } else {
        None
    }
}
```



Rust vs C++ (CI/CD)

Feature	C++	Rust
Build System	**Fragmented (Make, CMake, Ninja,etc.)**	**Unified: [cargo build]**
Cross Platform Builds	**No**	**Yes**
Crate Publishing	**No**	**Yes (crates.io)**
Version pinning	**No enforcements**	**Strictly versioned via cargo file**
Doc generation	**Not implicit (3rd party tools)**	**Yes, auto generated (rustdoc)**
Testing documentation	**No**	**Yes, automatic via cargo test**
Hosted documentation	**No (manually hosted / updated)**	**Yes, on docs.rs**
Unit testing	**Not implicit** (only via 3rd party tools)	**Yes, implicit, via #[test]**
Code coverage	**Not implicit** (only via 3rd party tools)	**Yes, implicit, via cargo tarpaulin**
Benchmarking	**Not implicit** (only via 3rd party tools)	**Built-in but nightly #[bench]**
Linter support / Static analysis	**3rd party tools** (clang-tidy, cppcheck)	**Yes, implicit, via cargo clippy**
Code formatting	**clang-format** (needs config per project)	**Yes, implicit, via cargo fmt**

Optimizations

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let’s consider the following C/C++ code and its Rust equivalent:

C/C++

```
void foo(const unsigned int * input, unsigned int * output) {
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

Rust

```
pub fn foo(input: &u32, output: &mut u32) {
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Where:

- rdi = input
- rsi = output

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let's consider the following C/C++ code and its Rust equivalent:

Why do we have this difference ?

C/C++

```
void foo(const unsig
    *output += *inpu
    *output += inpu
}
```

Rust

```
u32) {
```

Notice that we have used the **+=** operator. This means that the compiler first needs to read the value from the pointer ***output***, then add to that value the value from pointer ***input***, and finally write the new value into the ***output*** pointer.

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let's consider the following C/C++ code and its Rust equivalent:

Why do we have this difference ?

C/C++

```
void foo(const unsigned int *
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

Rust

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

However, *there is no guarantee* that the **output** pointer can't be access from a different thread. As such, the compiler has to write the new value to **output** pointer so that if another thread is trying to read it, it will read a correct value.
This also means that it has to perform a similar write for the second operation !

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let's consider the following C/C++ code and its Rust equivalent:

On the other hand, Rust knows that since output is a mutable reference, there is **only one such reference** and no other thread can access it. Because of this, it does not have to write the value after the first call to `+=` operator. Furthermore, since there **can not be mutable reference towards the input** variable (as there is already an immutable one), it can reuse its value for the second operation.

```
C/C
voi
}
```

```
output: &mut u32) {
nput;
```

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Map comparation between C++ and Rust

C++ vs Rust (on maps)

Let’s compare how various types of maps work on Rust and C++.

For this we will use:

- std::map (C++)
- std::unordered_map (C++)
- HashMap (Rust)
- BTreeMap (Rust)

The same algorithm will be written in both Rust and C++ and tested in Debug and Release mode. We will use GetTickCount API to measure time. Each variation of the build will be executed for 10 times and the average will be compute.

C++ vs Rust (on maps)

So … lets see the testing algorithm:

Rust

```rust
extern "system" { fn GetTickCount64() -> u64; }
fn get_time() -> u64 { unsafe { GetTickCount64() } }

use std::collections::{BTreeMap, HashMap};

#[derive(Debug, Copy, Clone)]
struct Test { v1: u64, v2: f32, v3: bool }

fn main() {
    let mut m: HashMap<u32, Test> = HashMap::new();
    let start = get_time();
    for i in 0..1_000_000 {
        let t = Test { v1: i as u64, v2: 1.5,v3: i%2==0};
        m.insert(i, t);
    }
    let end = get_time();
    println!("{}", end - start);
}
```

C++

```cpp
#include <Windows.h>
#include <map>
#include <unordered_map>
struct Test {
    unsigned long long v1;
    float v2;
    bool v3;
};
void main() {
    std::unordered_map<unsigned int, Test> m;
    auto start = GetTickCount64();
    for (auto i = 0; i < 1000000; i++) {
        m[i] = Test{ (unsigned long long)i,
                     1.5,i % 2 == 0 };
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```

C++ vs Rust (on maps)

So … lets see the testing algorithm:

Rust

```rust
extern "system" { fn GetTickCount64() -> u64; }
fn get_time() -> u64 { unsafe { GetTickCount64() } }

use s

#[der
struc

fn main() {
    let mut m: HashMap<u32, Test> = HashMap::new();
    let start = get_time();
    for i in 0..1_000_000 {
        let t = Test { v1: i as u64, v2: 1.5,v3: i%2==0};
        m.insert(i, t);
    }
    let end = get_time();
    println!("{}", end - start);
}
```

We will run the same algorithm using:

- HashMap
- BTreeMap

C++

```cpp
#include <Windows.h>
#include <map>
#include <unordered_map>
struc
    u
    f
    b
};
void main() {
    std::unordered_map<unsigned int, Test> m;
    auto start = GetTickCount64();
    for (auto i = 0; i < 1000000; i++) {
        m[i] = Test{ (unsigned long long)i,
                     1.5,i % 2 == 0 };
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```

We will run the same algorithm using:

- std::unordered_map
- std::map

C++ vs Rust (on maps)

So … lets see the testing algorithm:

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug) std:unordered_map	938	1266	1390	1328	1250	1297	1250	1344	1485	1437	**1298**
C++ (Release) std:unordered_map	297	234	235	281	266	266	234	235	234	234	**251**
C++ (Debug) std::map	1312	1765	1953	1875	1875	1859	1813	1812	1797	1828	**1788**
C++ (Release) std::map	156	141	172	157	141	171	172	156	172	156	**159**
Rust (Debug) HashMap	1141	1297	1265	1250	1281	1312	1359	1297	1343	1297	**1284**
Rust (Release) HashMap	78	78	63	78	78	94	93	94	94	94	**84**
Rust (Debug) BTreeMap	2703	3156	3078	2906	2765	2875	2937	2844	2860	2781	**2890**
Rust (Release) BTreeMap	93	93	109	110	125	110	125	141	125	125	**115**

C++ vs Rust (on maps)

The general conclusion after these tests is:

- Rust is slower the C++ when it comes to debug mode (due to many checks)
- In terms of Release mode, Rust is faster (however, it should be noted that we are not comparing the same algorithms and as such these tests might NOT be correct). However, since we've compared the standard algorithms from each (Rust and C++) libraries, the results are however relevant.
- The tests were performed on Windows 11 (using Microsoft compiler). To produce accurate results, other C++ compilers (such as clang and gcc) should be tested as well.

Vector comparation between C++ and Rust

Vectors

Let’s compare how efficient vector push method is for both C++ and Rust.

Rust

```rust
extern "system" {
        fn GetTickCount64 () -> u64;
}
fn get_time () -> u64 {
        unsafe { GetTickCount64() }
}
#[derive(Debug,Copy,Clone)]
struct Test { v1: i32, v2: f32, v3: char, v4: [u8;256] }
fn main() {
    let mut v: Vec<Test> = Vec::new();
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let start = get_time();
    for i in 0..10_000_000 {
        v.push(t);
    }
    let end = get_time();
    println!("{}",end-start);
}
```

C++

```cpp
#include <Windows.h>
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    auto start = GetTickCount64();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```

Vectors

Both codes were teste in the same environment, for 10 times and the average was recorded. All tests were run on x64 architecture (Debug and Release). Times are measures in milliseconds.
Keep in mind that **GetTickCount** function has an error margin of 16ms.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug)	2562	2562	2640	2578	2578	2515	2562	2578	2562	2547	**2568**
Rust (Debug)	1922	1844	1860	1843	1812	1813	1797	1781	1828	1813	**1831**
C++ (Release)	1828	1781	1797	1781	1781	1781	1797	1797	1821	1797	**1796**
Rust (Release)	1750	1750	1688	1719	1687	1703	1719	1687	1703	1687	**1709**

Vectors

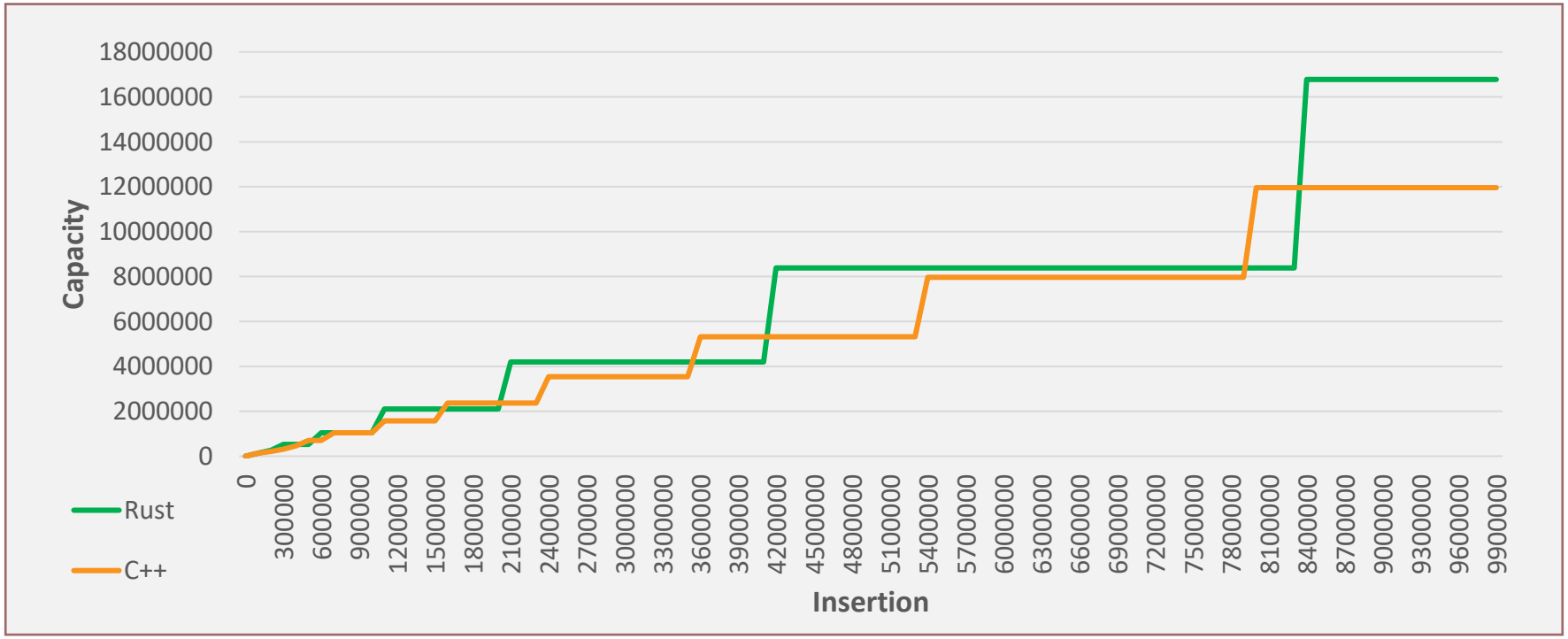
As a general conclusion, when it comes to vectors (and copying object not moving them), Rust is faster than C/C++ (in both debug and release modes).

We should point out that the build that was tested for C++ was compiled with Microsoft compiler (cl.exe) and it does not reflect results for gcc or clang (that might optimize the C++ code in a different way).

However, the question still remains on what's different in Rust vs C++ in terms of how vector works ?

Vectors

So … the difference lies in how growth algorithm works for those two cases (Rust and C++).



Vectors

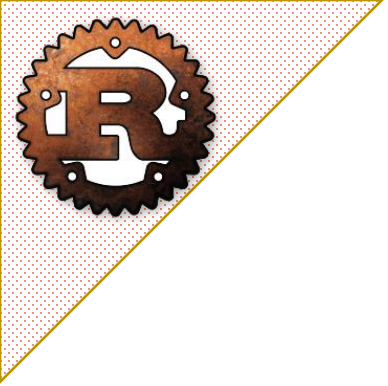
So … lets see the behavior if we reserve the memory from the start.

Rust

```rust
extern "system" {
         fn GetTickCount64 () -> u64;
}
fn get_time () -> u64 {
         unsafe { GetTickCount64() }
}
#[derive(Debug,Copy,Clone)]
struct Test { v1: i32, v2: f32, v3: char, v4: [u8;256] }
fn main() {
    let mut v: Vec<Test> = Vec::with_capacity(10_000_000);
    let t = Test{v1:5,v2:1.3,v3:'A',v4:[48u8;256]};
    let start = get_time();
    for i in 0..10_000_000 {
        v.push(t);
    }
    let end = get_time();
    println!("{}",end-start);
}
```

C++

```cpp
#include <Windows.h>
#include <vector>
struct Test {
    int v1;
    float v2;
    char32_t v3;
    uint8_t v4[256];
};
void main() {
    std::vector<Test> v;
    Test t;
    v.reserve(10000000);
    auto start = GetTickCount64();
    for (auto i = 0; i < 10000000; i++) {
        v.push_back(t);
    }
    auto end = GetTickCount64();
    printf("%d", (int)(end - start));
}
```



Vectors

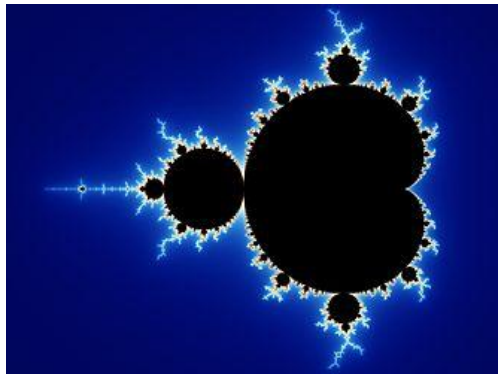
Test were performed in a similar manner like the previous ones (Debug and Release, 10 iterations and we compute the average).

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Average
C++ (Debug)	782	766	781	766	781	766	797	797	828	781	**784**
Rust (Debug)	984	1094	1063	1031	875	1032	1016	860	906	859	**972**
C++ (Release)	547	532	531	515	500	515	516	500	531	531	**521**
Rust (Release)	532	500	531	516	562	531	547	547	547	515	**532**

Keep in mind that there is an error margin of 16 ms for GetTickCount API. This means that the difference between C++ and Rust is insignificant (we can consider both at the same level).

Other 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:

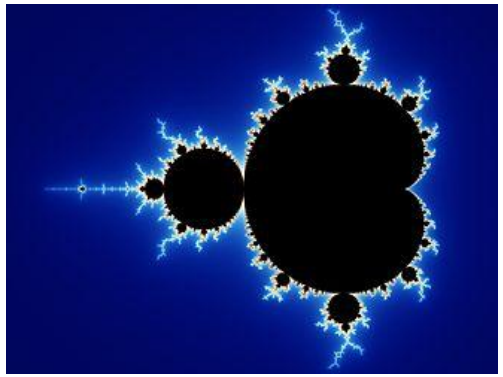
- Raw **CPU performance**
- **Loop optimization**
- **Floating-point performance**
- **Parallelism or threading** efficiency
- Compiler **code generation quality**

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

Other 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.



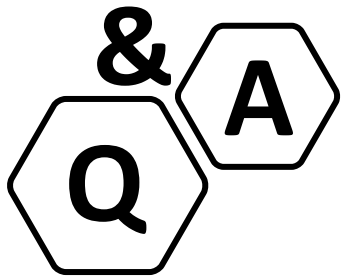
What It Measures:

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

#	Language	Time (sec)	Memory (MB)	#	Language	Time (sec)	Memory (MB)
1	**Rust**	0.95	35,598	14	**Dart**	4.29	45,175
2	**Chapel**	1.18	42,156	15	**Haskell**	6.64	51,057
3	**Julia**	1.54	357,138	16	**F#**	7.17	49,979
4	**C gcc**	1.64	35,582	17	**Swift**	7.27	49,312
5	**C++ g++**	2.36	38,281	18	**OCaml**	7.60	64,643
6	**Intel Fortran**	2.72	85,975	19	**Erlang**	53.86	98,140
7	**Go**	3.77	37,970	20	**PHP**	68.29	53,531
8	**Free Pascal**	3.91	35,529	21	**Ruby**	143.13	118,436
9	**Java**	3.96	58,348	22	**Lua**	159.01	652,796
10	**Ada 2012**	4.01	41,099	23	**Python 3**	182.94	62,173
11	**C#**	4.02	40,743	24	**PHP**	258.19	16,437
12	**Node.js**	4.05	144,757	25	**Smalltalk**	>5 min	175,542
13	**Lisp**	4.20	60,654	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)


&
Q
A