Rust vs C++

Igor Sadchenko

igor.sadchenko@gmail.com

Yauheni Akhotnikau, Alexander Zaitsev, Vitaly Shukela

About Rust

- **Rust** is a systems programming language sponsored by Mozilla Research, which describes it as a "safe, concurrent, practical language“, supporting functional and imperative-procedural paradigms.

- **Rust** is syntactically similar to C++, but its designers intend it to provide better memory safety while still maintaining performance.

- **Rust** grew out of a personal project started in 2006 by Mozilla employee Graydon Hoare. Mozilla began sponsoring the project in 2009 and announced it in 2010.

- **Rust** is an open source programming language.

About Rust

Syntax

- The concrete syntax of Rust is similar to C and C++, with blocks of code delimited by curly brackets, and control flow keywords such as if, else, while, and for.
- Despite the superficial resemblance to C and C++, the syntax of Rust in a deeper sense is closer to that of the ML family of languages.

Memory safety

- The system is designed to be memory safe, and it does not permit null pointers, dangling pointers, or data races in safe code.
- Rust core library provides an option type, which can be used to test if a pointer has Some value or None.
- Rust also introduces additional syntax to manage lifetimes, and the compiler reasons about these through its *borrow checker*.

Memory management

- Rust does not use an automated garbage collection system like those used by Go, Java or .NET Framework.
- Resources are managed through resource acquisition is initialization (RAII).
- Rust also favors stack allocation of values and does not perform implicit boxing.

About Rust

Ownership

- Rust has an ownership system where all values have a unique owner where the scope of the value is the same as the scope of the owner.
- Values can be passed by immutable reference using *&T*, by mutable reference using *&mut T* or by value using T.
- At all times, there can either be multiple immutable references or one mutable reference
- The Rust *compiler* enforces these rules at compile time and also checks that all references are valid.

Types and polymorphism

- The type system supports a mechanism similar to type classes, called "traits", inspired directly by the Haskell language.
- The implementation of Rust generics is similar to the typical implementation of C++ templates
- The object system within Rust is based around implementations, traits and structured types.

Cargo - the Rust package manager

```
Some common cargo commands are:

build       Compile the current project
doc         Build this project's and its dependencies' documentation
New         Create a new cargo project
run         Build and execute src/main.rs  Run the
test        tests
bench       Run the benchmarks
update      Update dependencies listed in Cargo.lock  Search
search      registry for crates
publish     Package and upload this project to the registry  Install a Rust
install     binary
```

Who use Rust

Atlassian : We use Rust in a service for analyzing petabytes of source code.

Coursera : Programming Assignments in secured Docker containers.

Mozilla : Building the Servo browser engine, integrating into Firefox, other projects.

Dropbox : Optimizing cloud file-storage.

SmartThings : Memory-safe embedded applications on our SmartThings Hub and supporting services in the cloud.

npm, Inc : Replacing C and rewriting performance-critical bottlenecks in the registry service architecture.

Classification

Rust

more control,
more safety

Classification

	Type safety	Type expression	Type checking	Garbage Collector
C	unsafe	explicit	static	No
C++	unsafe	explicit	static	No
Rust	safe	implicit/ explicit	static/ dynamic	No*

Classification

C++

+ Speed, no overhead
+ Community?
+ Commitee
- Missing package manager, safety

Rust

+ Package manager, Community
+ Tooling, Documentation, Concurrency
+ Speed, no overhead?
+ RFC process
- Syntax?
- Borrow checker;)
- Packages

Primitives

```rust
fn main() {
  //integers
  let i: i8 = 1; // i16, i32, i64, and i are available
  //unsigned
  let u: u8 = 2; // u16, u32, u64, and u are available
  //floats
  let f: f32 = 1.0; // f64 also available
  //booleans
  let b: bool = true; // false also available, duh
  //string and characters
  let c: char = 'a';
  let s: &str = "hello world";
}
```

Variable bindings

```
fn main() {
  let x: i32 = 1; //explicitly declare type
  let y = 2; //type inference
  let (a, b, c) = (1, 2, 3); //variable declaration via patterns
  let v = [1, 2, 3]; //array literals
  let s = "hello"; //string literal
  println!("*_^ x = {}, y = {}, a,b,c = {},{},{}", x, y, a, b, c);
}
```

Variable mutability

```rust
fn main() {
  let x: i32 = 1;
  x = 10;

  println!("The value of x is {}", x);
}

//cargo run
//error[E0384]: cannot assign twice to immutable variable `x`
// x = 10;
// ^^^^^^ cannot assign twice to immutable variable
```

Variable mutability

```rust
fn main() {
  Let mut x: i32 = 1;
  x = 10;

  println!("The value of x is {}", x);
}

//warning: value assigned to `x` is never read
// let mut x: i32 = 1;
// ^^^^^
//= note: #[warn(unused_assignments)] on by default
```

stack vs. heap

```rust
fn main() {
  let y: i32 = 1; //allocated on the stack
  let x: Box<i32> = Box::new(10); //allocated on the heap

  println!("Heap {}, Stack {}", x, y);
}

//cargo run
// Heap 10, Stack 1
// ^^^^^
```

Owning de-allocation

```rust
fn main() {
  let mut x: Box<i32> = Box::new(10); //allocated on the heap
  *x = 11;

  println!("The Heaps new value is {}", x);

//Scope for x ends here so the compiler adds the de-allocation
//free(x);
}

//cargo run
// The Heaps new value is 11
```

Borrowing

```rust
fn main() {
  // x is the owner of the integer, which is memory on the stack.
  let x = 5;
  // you may lend that resource to as many borrowers as you like
  let y = &x;
  let z = &x;
  // functions can borrow too!
  foo(&x);
  // we can do this many times!
  let a = &x;
}
```

Transfer ownership

```rust
fn main() {
  let x: Box<i32> = Box::new(5);
  let y = add_one(x);

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

fn add_one(mut num: Box<i32>) -> Box<i32> {
  *num += 1;
  num
}

//> cargo run
//> 6
//
```

Transfer ownership back

```
fn main() {
    let mut x: Box<i32> = Box::new(5);
    x = add_one(x);

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

fn add_one(mut num: Box<i32>) -> Box<i32> {
    *num += 1;
    num
}

//> cargo run
//> 6
//
```

Pattern matching

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

    match x {
        1 => println!("one"),
        2 => println!("two"),
        3 => println!("three"),
        4 => println!("four"),
        5 => println!("five"),
        _ => println!("something else"),
    }
}
```

Pattern matching

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

    match x {
        1 => println!("one"),
        2 => println!("two"),
        3 => println!("three"),
        4...7 => println!(“4 through 7"),
        _ => println!("something else"),
    }
}
```

Pattern matching

```rust
fn main() {
  struct Point {
    x: i32,
    y: i32,
  }
  let origin = Point { x: 0, y: 0 };
  match origin {
    Point { x: x, .. } => println!("x is {}", x),
  }
}
```

Iterators

```rust
fn main() {
    for i in (1..10).filter(|&x| x % 2 == 0) {
        Println!("{}", i);
    }
}

//> cargo run
//> 2
//> 4
//> 6
//> 8
```

Functions

```rust
//a function that takes and integer and returns an integer
fn add_one(x: i32) -> i32 {
    x + 1
}

fn main() {
    println!("1 plus 1 is {}", add_one(1));
}
```

```
//> cargo run
//> 1 plus 1 is 2
//
```

Closures

```
fn main() {
    Let add_one = |x| 1 + x;

    println!("The sum of 5 plus 1 is {}.", add_one(5));
}

//> cargo run
//> The sum of 5 plus 1 is 6.
//
```

Structs

```rust
#[derive(Debug)]
struct Point3d {
  x: i32,
  y: i32,
  z: i32,
}

fn main() {
  let mut point = Point3d { x: 0, y: 0, z: 0 };

  println!("The point is {:?}.", point);
}
//> cargo run
//> The point is Point3d { x: 0, y: 0, z: 0 }.
//
```

Traits

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Circle {
     fn area(&self) -> f64 {
         std::f64::consts::PI * (self.radius * self.radius)
     }
}
fn main() {
     let c = Circle {
          x: 0.0,
          y: 1.0,
          radius: 2.0,
     };
     println!("The circles's radious is {}", c.area());
}
//> cargo run
//> The circles's radious is 12.566370614359172
//
```

Traits with generics

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
fn main() {
    let c = Circle {
        x: 0.0,
        y: 1.0,
        radius: 2.0,
    };
    print_area(c);
}
```

Concurrency

```rust
use std::thread;
fn print_message(){
    println!("Hello from within a thread!");
}

fn main() {
    thread::spawn(print_message);
}
```

Tests

```rust
#[test]
#[should_panic]
fn adding_one() {
    let expected: i32 = 5;
    let actual: i32 = 4;

    assert_eq!(expected, actual);
}

//> cargo run
//> running 4 tests
//    test adding_one ... ok
//  test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Benchmark tests

```
#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}

//> cargo test
//> running 1 tests
//> test tests::bench_add_two ... bench:     1 ns/iter (+/- 0)
//> test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
//
```

Zero-cost abstraction

Preferring code duplication to abstraction due to high cost of virtual method calls

C++:

- Zero-cost abstraction mechanisms allow you to avoid runtime costs when possible.

Rust:

- The same!

Move semantics

Move constructors may leave objects in invalid and unspecified states and cause use-after-move errors

C++:

- Move constructors are suggested to leave the source object in a valid state (yet the object shouldn't be used in correct programs).

Rust:

- A built-in static analyzer disallows use of objects after they have been moved.
- The compiler can rely on this built-in analyzer for optimization.

Guaranteed memory safety

Use-after-free, double-free bugs, dangling pointers

C++:

- Smart pointers and references are preferred to raw pointers.
- Manual code review can spot use of raw pointers where smart pointers would suffice.

Rust:

- Smart pointers and references are preferred to raw pointers.
- Raw pointers can only be used inside unsafe blocks, which can automatically be found by tools.

Guaranteed memory safety

Null dereferencing errors

C++:

- References are preferred to pointers and cannot be null.
- Null dereferencing is still possible even for smart pointers, but is declared **as undefined** behavior and should never appear.
- Compilers assume that undefined behavior never happens, don't produce warnings, and use this for optimization (sometimes with fatal consequences for security).
- External static code analyzers can spot possible errors at compile time.

Rust:

- References are preferred to pointers and cannot be null.
- Null references can be emulated by **Option types**, which require **explicit** null checks before use.
- Smart pointers return **Optional references** and therefore require explicit checks as well.
- **Raw pointers** can be null, but they can only be used inside unsafe blocks. Unsafe blocks need to be carefully reviewed, but they can be found and marked automatically.

Guaranteed memory safety

Buffer overflow errors

C++:

- Explicitly coded wrapper classes enforce range checks.
- Debugging builds of the STL can perform range checks in standard containers.

Rust:

- All slice types enforce runtime range checks.
- Range checks are avoided by most common idioms (e.g. range-based for iterators).

Threads without data races

Data races (unsafe concurrent modification of data)
C++:

- Good programming discipline, knowledge, and careful review are required to avoid concurrency errors.
- External static code analyzers can spot some errors at compile time.
- External code sanitizers can spot some errors at runtime.
- Deadlocks?

Rust:

- The built-in borrow checker and Rust reference model detect and prohibit possible data races at compile time.
- Locking API makes it impossible to misuse mutexes unsafely (though still allowing *incorrect* usage).
- Deadlocks?

Object initialization

Uninitialized variables

C++:

- Constructors of user-defined types are recommended to initialize all object fields.
- Primitive types still have undefined values when not initialized explicitly.
- External static code analyzers can spot uninitialized variables.

Rust:

- All variables must be explicitly initialized before use (checked by the compiler).
- All types have defined default values that can be chosen instead of explicit initialization types.

Pattern matching

Forgetting to handle all possible branches in switch statements

C++:

- Code review and external static code analyzers can spot switch statements that don't cover all possible branches.

Rust:

- The compiler checks that match expressions explicitly handle all possible values for an expression.

Static (compile-time) polymorphism

Static interfaces for static polymorphism

C++:

- *Concepts* should provide this feature directly, but they've been in development since 2015 and are only scheduled for standardization in C++2x.
- Virtual functions and abstract classes may be used to declare interfaces.
- Virtual function calls may be optimized by particular compilers in known cases.
- Templates are second language.

```
template<uint32_t N>
struct S
{
    char arr[N];
};

int main()
{
    S<20> s;
}
```

Rust:

- Traits provide a unified way of specifying both static and dynamic interfaces.
- Static polymorphism is guaranteed to be resolved at compile time.

Type inference

Complex variable types become tedious to type manually

C++:

- The *auto* and *decltype* keywords provide type inference.

Rust:

- Local type inference (for a function body) allows you to explicitly specify types less frequently.
- Function declarations still require explicit types which ensures good readability of code.

Macroses

C++:

- #define

Rust:

- Macros by example
- Auto-derive

Standard library

Legacy design of utility types heavily used by standard library

C++:

- Structured types like std::pair, std::tuple and std::variant can replace ad-hoc structures.
- These types have inconvenient interfaces(though C++17 improves this).

Rust:

- Built-in composable structured types: tuples, structures, enumerations.
- Pattern matching allows convenient use of structured types like tuples and enumerations.
- The standard library fully embraces available pattern matching to provide easy-to-use interfaces.

Runtime environment

Embedded and bare-metal programming have high restrictions on runtime environment

C++:

- The C++ runtime is already fairly minimal, as it directly compiles to machine code and doesn't use garbage collection.
- C++ programs can be built without the standard library with disabled exceptions and dynamic type information, etc.

Rust:

- The Rust runtime is already fairly minimal as it directly compiles to machine code and doesn't use garbage collection.
- Rust programs can be built without the standard library with disabled range checks, etc.

Efficient C bindings

Using existing libraries written in C and other languages

C++:

- C libraries are immediately usable by C++ programs.
- Libraries in languages other than C++ require wrappers.
- Exporting a C interface requires only a simple *extern* declaration and pure procedural facade.
- There’s no overhead in calling C functions from C++ or calling C++ functions from C.

Rust:

- C libraries require Rust-specific header declarations.
- Libraries in languages other than Rust require wrappers.
- Exporting a C interface requires only a simple *extern* declaration and pure procedural facade.
- *There’s no overhead in calling C functions from Rust or calling Rust functions from C.

Compilation

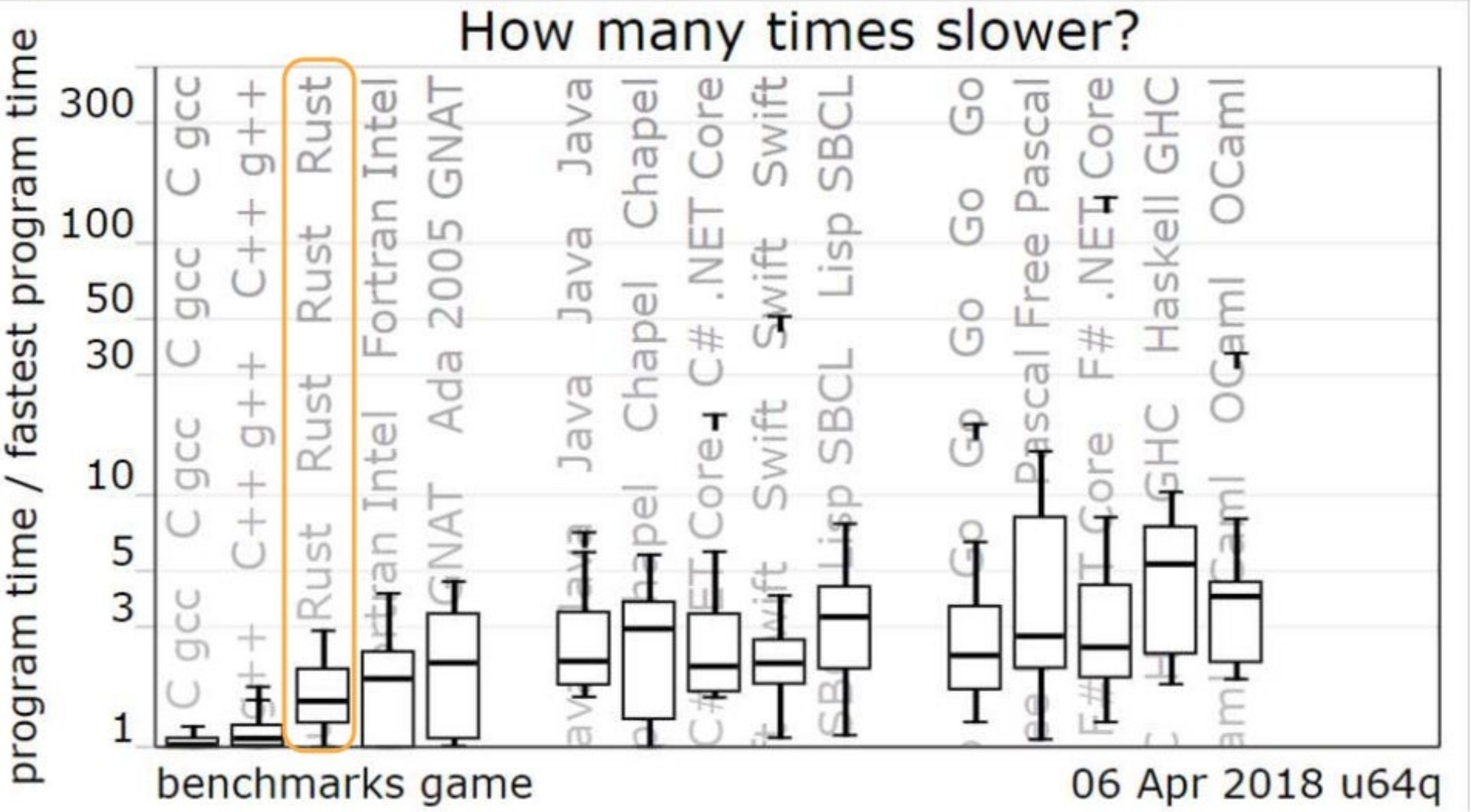
C++:

- Incremental compilation
- Separate, smaller code generation units

Rust:

- Incremental compilation
- Cross-compilation
- cargo check

Benchmarks



https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html

Summarize

- Does Rust have the right to life?
- Will Rust live long?
- Will Rust replace C/C++?
- Is Rust a worthy substitute for C/C++ in new projects?
- Will Rust become the main language for system programming?

Links

- Rust: http://rust-lang.org/
- Rust(rus): https://rustycrate.ru/
- Book: https://doc.rust-lang.org/stable/book/academic-research.html
- Cargo: https://crates.io/
- Rust by example: http://rustbyexample.com/

Many thanks!

Questions welcome :)

Igor Sadchenko igor.sadchenko@gmail.com +375 29 769-66-44