

An introduction to Rust language

V4 - Benoît Prieur - CC0
BarCamp Yerevan 
24-25 June 2023



Philosophy of Rust

- Integrate mature and experienced features from other languages
- Reason of the name “Rust”
- Mozilla Research: goal to invent a system language, **reliable**, **speed**, with simple and reliable **concurrency**



Rust as a Multi-paradigms Language

Rust as:

- Functional Language, evaluation and no-variable destruction
- Actor's Model (Concurrency)
- Procedural
- Object-Oriented



Improvements - Memory Management & Concurrency - Comparaisons with C++

- **Runtime** errors related to Memory Management (Allocation etc.) & Concurrency (Mutual Exclusion, Mutex etc.)
- Rust proposes to manage these aspects at **compiling**, goal to have a much reliable runtime



Memory Management

- Stack and Heap
- Type safety
- Memory Overflow at Runtime, IRL:
 - Ariane flight V88 (June 1996)



Usage: some examples

- Servo: HTML rendering (Firefox).
- Cargo: package management (Rust).
- Redox : Operating system Unix-like.
- Deno : new evolution of Node.js.
- Libra Core & Solana: crypto.
- Discord: messenger & VoIP.



Performance comparison, Rust, C++, C, Python etc.

Table 4. Normalized global results for Energy, Time, and Memory

- Rust
 - Energy 1.03
 - Time 1.04
 - Memory 1.34
- Python
 - Energy 75
 - Time 71
 - Memory 2.8
- <https://hal.science/hal-04083140>
- Microservices

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84



Package manager Cargo able to do...everything

- Command line

```
> cargo new --bin myproject
```

- Create **Cargo.toml** & **Main.rs**



Syntax quite close to C Language

- First example

```
let x = if n < 10 { n } else { n - 10 };
```

- Similar to C, conditions (if...else), loops (while) etc.



Generated Cargo.toml file

[package]

name = "myproject"

version = "0.1.0"

edition = "2021"

See more keys and their definitions at <https://doc.rust-lang.org/cargo/reference/manifest.html>

[dependencies]

chrono = "0.4"



Crates.io repository

- <https://crates.io/>
- <https://crates.io/crates/rand>

```
> cargo add rand
```

- The line: `rand = "0.8.5"`, added to `[dependencies]` in `Cargo.toml` file



Cargo commands

```
> cargo --version
```

```
> cargo new --bin [name]
```

```
> cargo new --lib [name]
```

```
> cargo clean
```

```
> cargo build
```

```
> cargo build --release
```

```
> cargo run
```

```
> cargo test
```



Immutable by default

- Mutable vs Immutable.
- Mastery of what you want to do exactly.
No guess by the compiler.



The *mut* keyword

```
fn main() {  
  
    let mut a = 5; // a is mutable  
  
    let b = a * 2; // b is not mutable  
  
    const c: u32 = 5; // constant  
  
    c = 3; // Error at compiling  
  
    b = 3; // Error at compiling  
  
    a = 2; // Ok at compiling  
  
    let a = a + 5; // Ok at compiling (shadowing)  
  
}
```



Ownership & borrowing

- A value always has a owner which is unique
- Compiling error otherwise
- Owner can be a variable, an object, a function, a data structure, a piece of code (loop etc.)



Ownership & Borrowship example

```
fn take_ownership(v: Vec<i32>) {  
    println!("{:?}", v);  
}  
  
fn main() {  
    let mut a = vec![1, 2, 3]; // a is the owner of the vector  
  
    let mut b = a; // b is the new owner  
  
    a.push(4); // error at compiling, a is not the owner  
  
    take_ownership(b);  
  
    b.push(5); // error at compiling, b is not the owner  
}
```




Immutable and Mutable references

- Something owned by an owner can have reference on it:
 - Immutable reference: **&**
 - Mutable reference: **& mut**
- Exclusively one way or the other



Enumeration & Filtering

```
enum Shape {  
    Point,  
    Rectangle(f64, f64),  
    Circle(f64),  
}
```



Enumeration & Filtering (II)

```
fn area(f: Shape) -> f64 {  
  match f {  
    Shape::Point => 0.0,  
    Shape::Circle(radius) => 3.14 * radius * radius,  
    Shape::Rectangle(a, b) => a * b,  
  }  
}
```



Enumeration & Filtering (III)

```
fn main() {  
    let point = Shape::Point;  
    let circle = Shape::Circle(2.0);  
    let rectangle = Shape::Rectangle(3.0, 4.0);  
    let area_point = area(point);  
    let area_circle = area(circle);  
    let area_rectangle = area(rectangle);  
}
```



Notion of Trait in Rust

- Equivalent to a C++ abstract class (kind of Interface)

// Define a trait named `Shape` with an `area` method

```
trait Shape {  
    fn area(&self) -> f64;  
}
```



Notion of Trait in Rust (II)

// Implement the `Shape` trait for a Circle struct

```
struct Circle {  
    radius: f64,  
}  
  
impl Shape for Circle {  
    fn area(&self) -> f64 {  
        3.14 * self.radius * self.radius  
    }  
}
```



Notion of Trait in Rust (III)

```
fn print_area(shape: &dyn Shape) {  
    println!("Area: {}", shape.area());  
}
```

```
fn main() {  
    let circle = Circle { radius: 2.0 };  
    print_area(&circle);  
}
```



Closure in Rust, an aspect of Functional Programming (lambda expression)

```
fn main() {  
    let add = |a, b| a + b;  
    let result = add(2, 3);  
    println!("Result: {}", result);  
}
```




Thank you :)