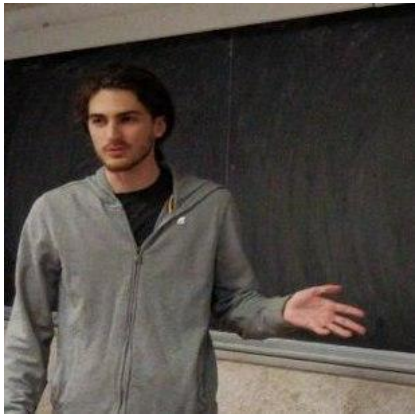
The Rust programming language

Who we are

Enrico Risa at OrientDB
Mozilla Tech Speaker
enrico.risa@gmail.com
@wolf4ood

Alessandro Cresto Miseroglio
Consultant Rust Developer Senior
alex179ohm@gmail.com
@alex179ohm



Giovanni De Luca
Student at Sapienza
and Mozilla Italia Volunteer
@gdl_jotaro_sama

Mozilla Italia

https://www.mozillaitalia.org

Rust Roma Meetup

https://www.meetup.com/it-IT/Rust-Roma/

Course Outline

1. Overview
2. Ownership & Borrowings
3. Data Structures
4. Std Library & Dependencies
5. Collections & Maps
6. Error Handling & Smart Pointer & Testing
7. Generics & Polymorphism
8. Concurrency

Lesson 1 Outline

- Overview
- Installation
- Syntax
- Data types
- Mutability
- Functions

Overview

Rust is a language empowering everyone to build reliable and efficient software.

Some info about Rust

Sponsored by **Mozilla Research**

The **0.1** release was in January **2012**

The **1.0.0** stable release was in May **2015**, since then the backward compatibility is guaranteed

Most 💕 programming language by StackOverflow survey: third place in 2015, **first place in 2016, 2017, 2018, 2019**!

Some info about Rust

- Multiparadigm
  - Imperative, functional, concurrent
- Static typing
  - With local type inference
- Compiled
- Zero Cost Abstraction
- Move semantics
- Guaranteed memory safety

Some info about Rust

- Threads without data races (cache friendly)
- Trait-based Generics
- Pattern matching
- Minimal runtime
- Efficient C binding
- Blazing fast

Rust goal

C C++ Java Ruby, Javascript, Python, ...

Control Safety

Rust goal

C C++ Java Ruby, Javascript, Python, ...

Control Safety

Rust

Rust is already in production!

Just to mention a few:

coursera

npm

Microsoft

CANONICAL

Google

Dropbox

ACADEMIA

SAMSUNG SmartThings™

system76

and many others, look at https://www.rust-lang.org/en-US/friends.html

Rust applications

Few of applications written completely in Rust:

Servo, a web browser engine by Mozilla

Async-std, a async framework, focus on easy-to-use fast networking and async io operations

Redox, full fledged Operating System, 100% Rust

Habitat, application automation framework

and many others, look at https://github.com/kud1ing/awesome-rust

Cargo

In Cargo, each library is called “**crate**”.

Stabilization pipeline for features is very quickly and nightly (as Rust language development itself).

“**Stability without stagnation**”

Cargo

Rust's package manager.

Manages dependencies and gives **reproducible builds**.

Cargo is one of the most powerful feature of Rust and it is the result of an awesome community!

Crates

Can be either a **binary** or a **library**.

libc, types and bindings to native C functions
xml-rs, an XML library in pure Rust
time, utilities for working with time-related functions
serde, a generic serialization/deserialization framework

... and more like **winapi**, **regex**, **url**, **rustc-serialize**, etc.

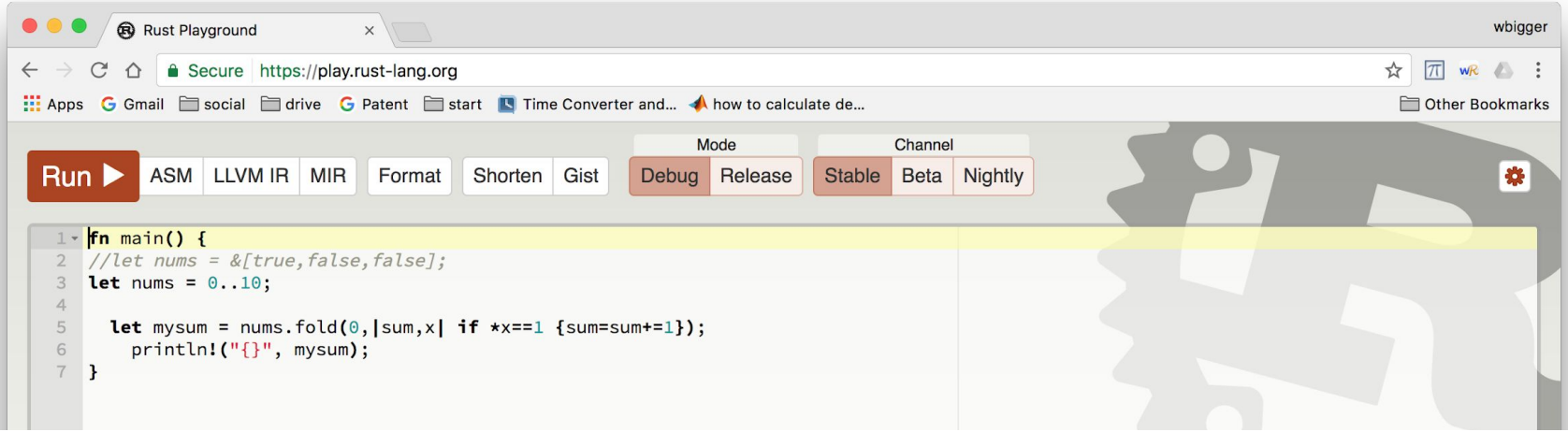
A full ecosystem

Formatter: **rustfmt**

Code completion: **rust-analyzer**

Linter: **clippy**

Playground



https://play.rust-lang.org/

Community

Community

Rust has an active and amazing community.
There are a lot of active channels, including:

forum, **reddit**, **IRC**, **youtube**, **twitter, blog, slack**

And initiative like:

- crate of the week
- rust weekly

Installation

Linux, *BSD, MacOSX

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Windows

You need:

Microsoft C++ build tools

Go to: https://visualstudio.microsoft.com/downloads/

Download **Community edition**

select **c++ tools** during the install

Download rust:

64 bit: https://win.rustup.rs/x86_64

32 bit: https://win.rustup.rs/i686

Syntax

Hello world!

```
fn main() {
    // Print text to the console
    println!("Hello World!");
}
```

Variables

```
let a = 5;
let b = "Rust Evangelism Strike Force";
let b: &str = "Rust Evangelism Strike Force";
let a = 34usize;
let c = 12_usize;
let c = 100_000;
```

Functions

```
fn add(a: i32, b: i32) → i32 {
  a + b
  // or
  // return a + b;
}
```

Control flow: **if**

```
if a < 2 {
  println!("a is less than 2");
}

if a < 2 {
  println!("a is less than 2");
} else {
  println!("a is equal or greater than 2");
}

let string = if a < 2 { "a is less than 2" } else { "is equal or greater than 2" };
println!("{}", string);
```

repetitions: **loop**

```
let mut a = 0;
loop {
  a += 1;
  print("{}", a);
  if a < 3 {
    continue
  }
  break
}
```

```rust
let mut a = 0;
let b = loop {
  a += 1;
  if a < 3 {
    continue
  }
  break 3
}
println!("{}", b); // prints 3
```

repetitions: **for**

```rust
for i in 0..3 {
  println!("{} ", i);
}
// output: 0 1 2

for i in 0..=3 {
  println!("{} ", i);
}
// output: 0 1 2 3
```

Don’t do that: for is a **syntactic sugar over loop**

```
let mut a = 0;
let b = for i in 1..=4 {
    a += i
};
println!("{:?}", b);
```

```rust
let mut a = 0;
let b = for i in 1..=4 {
    a += i
};
println!("{:?}", b); // output: ()
println!("{}", a); // output: 10
```

for is something like:

```
let mut a = 0;
let mut iter = [1, 2, 3, 4].into_iter();
let b = loop {
    if let Some(i) = iter.next() {
        a += i;
        continue;
    }
    break ();
};
println!("{:?}", b); // output: ()
println!("{}", a); // output: 10
```

repetitions: **while**

```
let mut a = 0;
while a < 5 {
  println!("{}", a);
  a += 1;
}
```

or you can do something like this

```
let mut iter = [1, 2, 3, 4].into_iter();
while let Some(i) = iter.next() {
  println!("{}", i);
}
```

Objects are declared as **Structs**

```
struct Object {
  field: i32,
}
```

Methods

```
impl Object {
  fn new() → Self {
    Object { field: 0 }
  }

  fn get_field(&self) → i32 {
    self.field
  }

  fn add_to_field(&mut self, arg: i32) → {
    self.field += arg;
  }
}
```

Primitive Data Types

booleans

Booleans can be true or false

```
let t = true;
let f = false;
```

integers

Different sizes differents types

i8, i16, i32, i64, u8, u16, u32, u64

the letters i and u denotes respectively signed or unsigned

isize, usize are architecture dependents (different architecture, different size)

char

Defined as “Unicode scalar value” can be intended as character value supporting UTF-32 (4 bytes length)

```
let a = 'a';
let keyboard = '⌨';
```

array

Fixed size length list of same type elements.

Declared:

let array: [type; N] = [elem, elem, elem, ...];

```
let array: [i32; 5] = [0, 1, 2, 3, 4];
```

slice

is array without the fixed size length, the size is resolved a compile time and must be referenced if declared or the size must be known at compile time.

let slice: &[type] = &[elem, elem, elem, ...];

```
let slice: &[i32] = &[1, 2, 3, 4];
```

str

It’s the most primitive string type

It’s like a “string slice”

```
let s: &str = "Rust Evangelism Strike Force";
```

tuple

A list of arbitrary type elements.

```
let tuple = ("Resf", 32i32, [7, 8, 9], ("Resf", 3.14));
```

Mutability

Variables have mutability property

The value of a variable can only be changed if the variable is declared as mutable

```
// this not compile
let x = 5;
println!("{}", x);
x = 6; // error: variable x must be declared mutable
println!("{}", x);
```

Declaring variable mutable is done by the mut keyword

```
let mut x = 5;
println!("{}", x);
x = 6;
println!("{}", x); // this outputs 6 as expected
```

Mutate variable's type is not allowed, even if the variable is declared as mutable

```rust
// this not compile
let mut x = "Rust Evangelism Strike Force";
println!("{}", x);
x = x.len(); // error: expected &str found usize
println!("{}", x);
```

const

Constants are immutable and they doesn't have mutability property, you cannot use mut with const

```
// this not compile
const mut STOCK_PRICE: f32 = 100.72;
// this is ok
const ONE_MILION: i32 = 1_000_000;
```

Functions

Simple function is declared with the fn keyword

```
fn print_hello() {
  println!("Hello Resf");
}

fn main() {
  print_hello();
}
```

Arguments must be declared with type

```
fn add_one(mut arg: i32) {
    arg += 1;
    println!("{}", arg);
}

fn main() {
  let arg = 3;
  add_one(arg); // output: 4
}
```

Function everytime returns

```rust
// return ()
fn hello() {
    println!("Hello Resf");
}

// return i32
fn six() -> i32 {
  6
}
```

Variadic arguments are just supported in unsafe C bindings

```
// this not compile
fn variadic(arg: i32, args: ...) {
    println!("Hello {}", arg);
}
```

Variadic workaround

Elements of differents types:

struct, tuple

Elements of same types:

struct, tuple, array, slice

Multiple returns is done via **tuple** or **struct**

```rust
fn multiple_returns() -> (i32, String) {
    (3, "Resf".to_string())
}

let (num, string) = multiple_returns();
println!("num = {}", num);
println!("string = {}", string);
```

```rust
struct Object(pub i32, pub String);

fn struct_return() -> Object {
  Object(3, "Resf".to_string())
}

let o = struct_return();
println!("num = {}", o.0);
println!("string = {}", o.1);
```

Next lesson

Giovedì 21 novembre orario 18-20

Per info e domande:

alex179ohm@gmail.com

enrico.risa@gmail.com

Oggetto: **corso rust sapienza**

Slack: http://rust-italia.herokuapp.com channel: #rust-roma