Rust

A short tutorial

Paweł Perechuda Wojciech Kamiński Jakub Kasprzak

Agenda

- Introduction - brief history, motivation etc.
- Writing program in Rust, basic programming concepts
- Good practices enforced by compiler, mechanism of borrowing
- OOP capabilities in Rust

History

- Started as a personal project of Mozilla Research employee Graydon Hoare in 2006,
- Successfully bootstrapped in 2011,
- First stable release (Rust 1.0) in 2015,
- Since 2021 developed by the Rust Foundation (Founders: AWS, Huawei, Google, Microsoft, Mozilla),

History pt. 2

- In december 2022 it became the first language other than C and assembly to be supported in the development of the Linux kernel,
- Supposedly MS Windows kernel is being rewritten in Rust (Win11 Insider builds seems to contain parts of kernel already rewritten in Rust).

What is Rust?

- Multiparadigm, general-purpose programming language,
- Keywords: performance, type safety, concurrency,
- Memory safety achieved by ensuring that all references used references are valid,
- Has pointers (used in `unsafe` code),
- No garbage collector!

Selected features

- Compiler driven development,
- Memory checks in compile time,
- Const/Immutable variables by default,
- Clear error output,
- Package manager similar to Python’s pip,
- Support for project structure by convention out of the box.

Meet Ferris

Getting started with rust

You can get rustup from rust-lang.org

Basic rust tools and components:

- rustup - manages rust and associated tools
- rustc - compiler
- cargo - build system and package manager
- rustfmt - official formatter
- clippy - official linter

Hello world

Create new project with `cargo new <project_name>`

Compile with `cargo build` or with `rustc <file.rs>`

main.rs

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

Cargo.toml

```toml
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[dependencies]
```

Primitive data types

Signed integers: `i8 i16 i32 i64 i128 isize`

Unsigned integers: `u8 u16 u32 u64 u128 usize`

Boolean: `bool`

Characters: `char`

Floating point: `f32 f64`

Tuples: `(i32, char, f64) (char, usize) …`

Arrays: `[i32; 4] [char; 8] …`

Variable mutability

```rust
fn main() {
    let number: i32 = 5;
    println!("{number}");
    number = 6;     cannot mutate immutable variable `number`
    println!("{number}");
}
```

```rust
fn main() {
    let mut number: i32 = 5;
    println!("{number}");
    number = 6;
    println!("{number}");
}
```

Constant vs immutable

```rust
fn main() {
    //immutable variable
    let r1: bool = rand::thread_rng().gen_bool(0.5);
    //const variable
    const r2: bool = rand::thread_rng().gen_bool(0.5);
    println!("{r1},{r2}");
}
```

Functions

```rust
fn main() {
    const FACTORIAL_10: u128 = factorial(10);
    println!("{FACTORIAL_10}");
}

const fn factorial(n: u128) -> u128 {
    if n == 0 {
        1
    } else {
        factorial(n - 1) * n
    }
}
```

Shadowing

```rust
fn main() {
    let mut number: String = String::new();
    std::io::stdin().read_line(buf: &mut number).unwrap();
    let number: u128 = number.trim().parse().unwrap();
    println!("{number}");
}
```

Flow control (branching)

match

```rust
fn main() {
    let text: &str = match 3 {
        0 => {
            zero_detected();
            "It is 0"
        }
        1 => "It is 1",
        _ => "It is something else",
    };
    println!("{text}");
}
```

If, else

```rust
fn main() {
    let number: i32 = 3;
    let text: &str = if number == 0 {
        zero_detected();
        "It is 0"
    } else if number == 1 {
        "It is 1"
    } else {
        "It is something else"
    };
    println!("{text}");
}
```

Flow control (repetitions)

loop

```rust
fn main() {
    loop {
        println!("Doctor Strange: Dormammu, I've come to bargain.");
        println!("Dormammu: *sneezes*");
        println!("Doctor Strange: 💀");
    }
}
```

while

```rust
fn main() {
    let arr: [i32; 2] = [1, 2];
    let mut i: usize = 0;
    while i < 2 {
        println!("{}", arr[i]);
        i += 1;
    }
}
```

for

```rust
fn main() {
    let tup: (i32, &str) = (0, "I like bread");
    for i: i32 in 0..2 {
        println!("{}", tup.i);    no field `i` on type `({integer}, &str)`
    }
}
```

Mechanism of borrowing

```c
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *string, *string_so_far;
    int i, length; length = 0;

    for(i=0; i<argc; i++)
    {
        length += strlen(argv[i])+1;
        string = malloc(length+1);

        /*  * Copy the string built so far. */
        if(string_so_far != (char *)0)
            strcpy(string, string_so_far);
        else *string = '\0';

        strcat(string, argv[i]);

        if(i < argc-1) strcat(string, " ");

        string_so_far = string;
    }

    printf("You entered: %s\n", string_so_far);
    return (0);
}
```

- Memory leaks in program to the left
- Ensuring memory Safety in rust is done via ownership of reference, so there is no multiple owners of the pointer, its prohibited in this language. When this is ensured,
  Compiler knows how to compile code in a way to automatically dealocate unused memory, its decided during compile time., since there is only one owner of memory its easy . Compiler calls “drop” operation which is equivalent of “delete” in cpp.

Taking ownership

```rust
fn take_ownership(value: String) {
    println!("Received value: {}", value);
}

fn main() {
    let my_string = String::from("Hello, world!");
    take_ownership( value: my_string);
     println!("My string: {}", my_string);
}
```

```
error[E0382]: borrow of moved value: `my_string`
  --> src\main.rs:47:32
   |
42 |     let my_string = String::from("Hello, world!");
   |         --------- move occurs because `my_string` has type `String`, which does not implement the `Copy` trait
43 |     take_ownership(my_string);
   |                    --------- value moved here
help: consider cloning the value if the performance cost is acceptable
   |
43 |     take_ownership(my_string.clone());
   |                             ++++++++
```

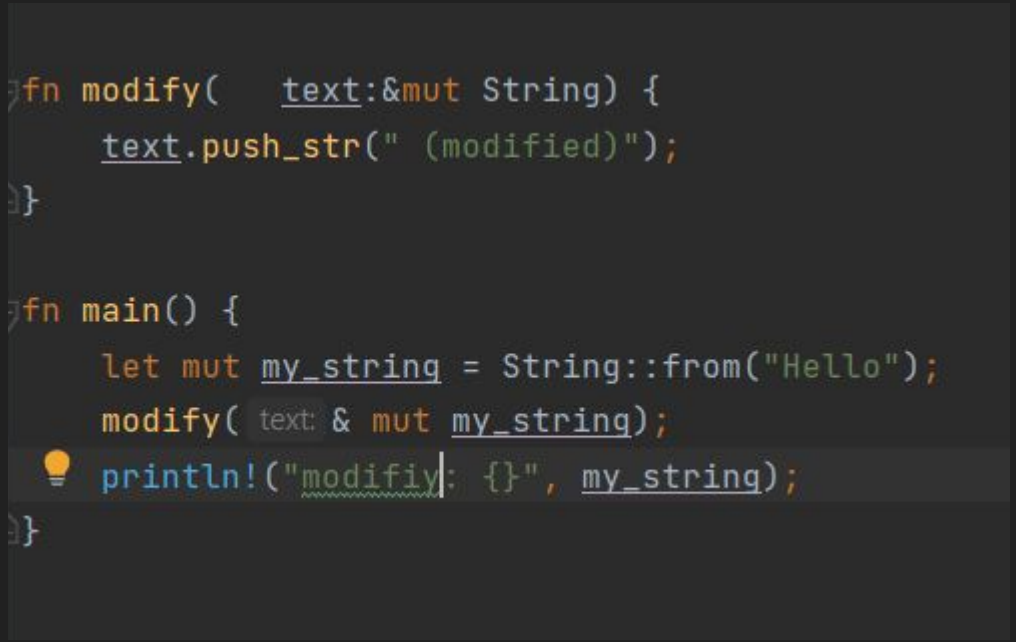
Borrowing a value, not mutable

```rust
fn modify_vector_values(vector: & Vec<i32>) {
    for value in vector.iter_mut() {
        *value *= 2; // Multiply each value by 2
    }
}
fn main() {
    let  numbers = vec![1, 2, 3, 4, 5];
    println!("Original vector: {:?}", numbers);
    modify_vector_values( vector: & numbers);
    println!("Modified vector: {:?}", numbers);
}
```

```
-a----          6/4/2023   5:48 PM            200 cargo.toml
   |                ^^^^^^^^^^^^^^^^^ `vector` is a `&` reference, so the data it refers to cannot be borrowed as mutable
   |
help: consider changing this to be a mutable reference
   |
18 | fn modify_vector_values(vector: &mut Vec<i32>) {
   |                                 ~~~~~~~~~~~~~~
```

Borrowing mutable reference

My task in tutorial requires you to borrow mutable reference, And modify vector ,than show that vector values were in fact modified.



Built in tuples

```
let tuple = ("A", "Wonderful", 44);

let first_item = tuple.0;
let second_item = tuple.1;
let third_item = tuple.2;

let (one, two, three) = tuple;
```

Classes (structs)

```
struct User {
    username: String,
    email: String,
    date_of_birth: LocalDate,
    last_login: LocalDate
}
```

Methods

```
impl User {
  fn get_reversed_username(self) -> String {
    self.username.chars().rev().collect()
  }
}
```

Consumption of self

- Self is moved to the method so you can call a method on this object only once

```
fn get_reversed_username(&self)
```

Why this won't work?

```
fn set_reversed_username(&self) {
  self.username = self.get_reversed_username();
}
```

Self is not mutable!

Constructors

It is impossible to define constructors in Rust!

Inheritance

Rust does not support inheritance!

Ways to create an object

- `Constructor` method

```
fn new(username: String) -> Self {
        Self {
            ...
        }
}
```

Ways to create an object pt. 2

```
let mut copied_user = User {
        username: "John".to_string(),
        ..new_user
};
```

Mind the move of some `new_user` fields!

Interfaces

- For clarity they’re named `trait`s in Rust

```
trait Summary {
    fn summarize(&self);
}
```

Interface implementation

Example implementation of `Display` trait

```
impl Display for User {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    write!(f, "{}\t{}", self.email, self.username)
  }
}
```

Default trait implementations

`derive` attribute allows new items to be automatically generated. Here's generated default impl of `Debug` trait!

```
#[derive(Debug)]
struct User {
    ...
}
```

Enums

```
enum PersonStatus {
    Ok,
    Ill(i128),
    Yes{no: Box<str>},
    Registered(String, i64, Mutex::<String>)
}
```

Matching enum values

```
if let PersonStatus::Ill(value) = status {
    println!("Person is ill! Val: {value}");
}
```

- `match` will also work similarly. A branch would be defined like that `PersonStatus::Ill(value) => {…},`