Rust #1: Motivation and Introduction

March 10, 2021

1 What are we doing here?

1.1 Why are we here?

We are here to learn a new way of thinking about complex computer systems with the help of the Rust programming language.

1.2 What this entails

We will have to do a few things in order to gain the most in the short amount of time we have:

- Listen and ask questions during the lecture
- Complete short homework exercises
- Work our way through a final group project
- Read/Watch/Listen/Do anything and everything we can lay our eyes upon!

1.3 Rust doesn't solve all of your problems

While Rust does solve most memory safety issues (more on this later), most bugs are still caused by tired and underslept programmers.
Your health is more important than your work.

1.4 Rust doesn't solve all of your problems

Developing more secure and memory-safe code in abstract sounds good. But it's not always.
Developing a secure drone strike system won't help anyone.
Secure face recognition system for the police won't help anyone either.
A secure app that serves as a tool to further underpay and undermine the vulnerable workforce will bring far more trouble if it's unhackable using simple memory exploits.

2 Motivation

2.1 Who is interested in Rust?

Rust is the most loved language for the fifth year in a row! (86.% of StackOverflow users)
Rust is used in production in an increasing number of companies: Mozilla, Atlassian, Microsoft, Google, Godot, 1Password, Dropbox, Zeplin, npm, Academia.edu, Sentry, Cloudflare, Coursera, Figma, Postmates etc.
But Rust's concepts will help in many other languages as well!

2.2 Why should you be interested in Rust?

Rust is an extremely interesting and relatively new language that can be used in a lot of domains, including, but not limited to:

- Web through WebAssembly
- Operating Systems
- Embedded Software
- Large Systems Software (Browsers, Servers, Databases etc.)
- Games, GUI, CLI applications too!

2.3 Resources

All of these can be found in our repository! And even more are available online!

3 A short history of systems programming

3.1 The origins of C

The C programming language appeared during Unix development in 1972.
Since it was created for a specific purpose and a specific computer, on the one hand it adapted to the needs of the programmers, and on the other it adopted a large amount of somewhat unique and unpopular ideas and concepts.

3.2 Possible solutions

C++ is born to help address some of these problems, introduces 'zero cost' abstractions, aimed at providing a nice interface for the programmer to use which compiles down to an almost ideal machine code.
Still has the old instruments, hangs on to C's machine model and tries to encourage using the new modern safe concepts, which are not ideal either

3.3 Modern ideas

In the meantime, languages like Java, Ruby and Python start sprawling up, presenting another model of growth - they are garbage-collected and are able to present even more complex abstractions (at the expense of the speed).
Go and others try to tackle C's speed and low-levelness, unsuccessfully.

4 Pitfalls of the old ways

4.1 Memory layout

```
int test()
{
        int b = 5;
        return b;
}

int main()
{
        int a = test();
        return a;
}
```

4.2 Memory layout

stack picture

4.3 Buffer overflows

```
int main()
{
        char s[100];
        int i;
        printf("\nEnter a string :  ");
        gets(s);

        return 0;
}
```

4.4 Null pointers

```
int test(int* pointer)
{
        *pointer = 5;
        return 0;
}

int main()
{
        char* text = malloc(10);
        *text = 'c';
        return 0;
}
```

4.5 Dangling pointers

```
int *test()
{
        int y=10;
        return &y;
}

int main()
{
        int *p = test();
        printf("%d", *p);
        return 0;
}
```

4.6 Invalidated iterators and double free

```
void main()
{
        Vec*vec=vec_new();
        vec_push(vec, 107);

        int* n = &vec->data[0];
        vec_push(vec,110);
        printf("%d\n", *n);

        free(vec->data);
        vec_free(vec);
}
```

4.7 No real error checking

```
int main()
{
        /*
         * Returns -1 for ERROR #1
         * Returns -2 for ERROR #2
         * .......
         * */
}
```

4.8 And many more...

- Memory leak - you can forget to free data
- Thread unsafety - another function can be modifying the same memory

5 Where and why is Rust better?

5.1 What is Rust?

A modern systems programming language.
First stable version in 2014.
Gets rid of unnecessary old ideas, combining them with some of the fresh concepts.

5.2 Almost everything makes sense

It does not care about C's old ways from the 70s which have been kept up in many languages and systems since.
It does not try to needlessly attach new stupid things to it. Easily gets rid of bad ideas since it's a young language.

5.3 Memory safety guarantees

Rust guarantees, at compile time, that programs are memory-safe.
This means:

- No buffer overflows
- Everything is bounds-checked (e.g. no null pointers, no dangling pointers)
- No data races (general thread-safety)
- No memory is ever written to by two things at a time

- No memory is ever written and read at the same time

5.4 Tradeoffs

These guarantees don't come without a cost.
Some of them include:

- Relatively long compile time
- You sometimes have to fight with the compiler (It's always right though, and even tries to help)
- A whole new different approach to things
- It's a young language! (This also means you can help)

5.5 Improvements in almost every field

Rust does not only get rid of the problems of C and garbage-collected languages.
It tries to be better at a lot of other things:

- Super great error handling!
- Algebraic data types! (Functional programming concept)
- Advanced macro processing!
- Speed increase from C since programs often know what to expect!

5.6 An amazing ecosystem

Rust has a great ecosystem:

- rustc compiler is super helpful, and has great IDE integrations!
- cargo is a unified standard for:
  - Package management (pip, yay)
  - Dependency resolving (setup.py, requirements.txt)
  - Project management (makefile, CMake)
  - Testing (gtest etc.)
  - Documentation (pandoc, asciidoc etc.)
- rustfmt is a standard formatter
- clippy is a standard linter

6 Basic syntax and ecosystem

6.1 Hello world!

All of the code snippets for this lecture are available in the repository!

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

```
$ rustc main.rs
$ ./main
Hello, world!
```

6.2 Hello cargo!

```
$ cargo new hello_cargo
$ cd hello_cargo
$ ls
$ cat Cargo.toml
$ cat src/main.rs

$ cargo build
$ target/debug/hello_cargo

$ cargo run
```

7 Ownership and lifetimes

7.1 Ownership and lifetimes

7.1.1 Abstract

How would a safe system look like?
Let's go over a few main points:

- There can be many readers with no writers at the same time
- There can be only one writer with no readers at the same time
- Values can be used only as long as they still exist

7.2 Ownership and lifetimes

7.2.1 General

In Rust's terminology, there are two notions of borrowing:

- Shared reference &
- Mutable reference &mut

Rust forces us to abide by the following principles:

- Every value has a single owner at any given time. (You can move a value from one owner to another, but when value's owner goes away, the value goes away with it too)
- You can borrow a reference to a value, for as long as the reference doesn't outlive the value. (Borrowed references are temporary pointers, they allow you to operate with values you don't own)
- You can only modify a value when you have exclusive access to it.

7.3 Ownership and lifetimes

7.3.1 General

The lifetime of a value starts when it's created and ends the last time it's used.
Rust computes lifetimes at compile time, we don't have to do any allocations and frees, Rust borrows and drops memory for us.

7.4 Ownership and lifetimes

7.4.1 Examples

Let's see how Rust automatically drops any values after their scope ends (and their owner dies):

```
fn main() {
    let a: u32 = 5;
    let b = Box::new(5i32);

    {
        let a: u32 = 5;
        let b = Box::new(5i32);
    }
}
```

Rust handles all data this way, no matter the complexity, it will clean up file handlers, network sockets, vectors etc.

7.5 Ownership and lifetimes

7.5.1 Examples

Let's see how Rust handles Strings. Does this compile?

```
fn main() {
    let s: String = "text".to_string();
    let u = s;
    println!("{}", s);
}
```

7.6 Ownership and lifetimes

7.6.1 Examples

Does this one compile?

```
fn main() {
    let s: String = "text".to_string();
    let u = s;
    println!("{}", u);
}
```

7.7 Ownership and lifetimes

7.7.1 Examples

What is different for u32?

```
fn main() {
    let s: u32 = 42;
    let u = s;
    println!("{}", s);
}
```

7.8 Ownership and lifetimes

7.8.1 Copy and Move

Rust has a special notion which helps it figure out whether the value should be copied or moved.
We'll discuss this a bit later, but for now you can just remember that Rust copies stack-allocated primitive values, and moves ownership of heap- allocated values.

7.9 Ownership and lifetimes

7.9.1 Examples

```
fn om_nom_nom(s: String)
{
    println!("I have consumed {}", s);
}

fn main()
{
    let s: String = "text".to_string();
    om_nom_nom(s);
    println!("{}", s);
}
```

7.10 Ownership and lifetimes

7.10.1 Examples

```
fn om_nom_nom(n: u32)
{
    println!("{} is a very nice number", n);
}

fn main()
{
    let n: u32 = 110;
    let m = n;
    om_nom_nom(n);
    om_nom_nom(m);
    println!("{}", m + n);
}
```

7.11 Ownership and lifetimes

7.11.1 Examples

Let's see how Rust will handle complex correct code:

```
fn borrow(text: &String) {
    println!("I immutably borrowed this text: {}", text)
}

fn mut_borrow(text: &mut String) {
    text.push_str(" + new text");
    println!("I mutably borrowed text and changed it to:
}

fn main() {
    let mut og_text = "old text".to_string();

    borrow(&og_text);
    borrow(&og_text);
    mut_borrow(&mut og_text);
}
```

7.12 Ownership and lifetimes

7.12.1 Examples

Let's try to invalidate an iterator:

```rust
pub fn main() {
    let mut data = vec![1, 2, 3];
    let x = &data[0];

    data.push(4);

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

7.13 Ownership and lifetimes

7.13.1 Examples

Let's try to return a dangling pointer:

```rust
fn as_str(data: &u32) -> &str {
    let s = format!("{}", data);

    &s
}

pub fn main() {
    let n = 5;
    as_str(&n);
}
```

7.14 Ownership and lifetimes

We've only looked at some of the simplest cases, Rust's system is a lot more complex. We'll go over lifetimes more in the future.
Since the concept of borrow checker is unique to Rust, until you will get used to it you will often encounter compiler errors when working.
The compiler becomes better and better at working with lifetimes, and provides super helpful errors, but it still can be stressful to fight with it sometimes.

8 A simple practice project

8.1 Guessing game

8.1.1 Basics

```rust
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = 5;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

8.2 Guessing game

8.2.1 Cargo dependencies

Let's add a random number generator. We have to add a dependency for rand crate for this. (crates are Rust's libraries and modules)
Add this to your Cargo.toml file:

```
[dependencies]
rand = "0.8.3"
```

8.3 Guessing game

8.3.1 Random number generation

```rust
use rand::Rng;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

8.4 Guessing game

8.4.1 (Wrong) Input comparison

```rust
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // --snip--

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

8.5 Guessing game

8.5.1 Input comparison

```rust
    // --snip--

    let mut guess = String::new();

    io::stdin().read_line(&mut guess).expect("Failed to
```

```
    let guess: u32 = guess.trim().parse().expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

8.6 Guessing game

8.6.1 Looping

```
    // --snip--

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

8.7 Guessing game

8.7.1 Breaking out of the loop

```
        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

8.8 Guessing game

8.8.1 Handling invalid input

```
        // --snip--

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        // --snip--
```

8.9 More on cargo

```
$ cargo build
$ cargo run
$ cargo new

# Building a release binary:
# (Takes more time, optimizes more)
$ cargo build --release
$ target/release/main

# Generate the documentation of the dependencies:
$ cargo doc --open

# Running unit tests and integration tests:
$ cargo test
```