APPS@UCU

Rust #1: Motivation and Introduction

Sultanov Andriy

APPLIED SCIENCES FACULTY

Contents

1. A short history of systems programming
2. Pitfalls of the old ways
3. Where and why is Rust better?
4. Basic syntax and concepts
5. The Rust ecosystem and a little more

A short history of systems programming

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.

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.

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.

Pitfalls of the old ways

General stuff

General stuff about how they are not memory-safe.
Should probably spend some time explaining the memory layout, so it'd be possible to explain buffer overflows, dangling pointers and null pointers.

Buffer overflows

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

        return 0;
}
```

Null pointers

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

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

Dangling pointers

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

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

Invalidated iterators

```
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);
}
```

No real error checking

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

And many more...

- Memory leak - you can forget to free data
- Thread unsafety - another function can be modifying the same memory
- Double free - you can free the same memory twice (as a part of a struct, for example)

Where and why is Rust better?

What is Rust?

Safe, Fast, Easy to write. Choose three

A modern system programming language.

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

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.

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

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)

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!

An amazing ecosystem

Modern languages are not only language specifications. Their use almost always depends on compilers, package management systems, documentation, formatters, and test suites!

And 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

Basic syntax and concepts

Hello world!

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

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

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
```

The Rust ecosystem and a little more...

Guessing game
Basics

Type system explanation, print macros, expect() and methods

```
use std::io;

fn main() {
    println!("Guess the 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);
}
```

*Explain how dependencies work.
Or at least try to.*

Add this to your Cargo.toml file:

```
[dependencies]
rand = "0.5.5"
```

Guessing game
Random number generation

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

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);
}
```

Guessing game
(Wrong) Input comparison

Match expression, enumerators explanation.

```
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!"),
    }
}
```

Guessing game
Input comparison

Ownership explanation, maybe add a slide about that?

```
    // --snip--

    let mut guess = String::new();

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

    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!"),
    }
}
```

Ownership and lifetimes
General

Rust enforces its own (rational) system of ownership and lifetimes. A few main points:

- Borrowing
  - We can have multiple shared (immutable) references at once (with no mutable references) to a value.
  - We can have only one mutable reference at once (no shared references to it)
- Lifetimes
  - The lifetime of a value starts when it's created and ends the last time it's used
  - Rust doesn't let you have a reference to a value that lasts longer than the value's lifetime
  - Rust computes lifetimes at compile time, there are no allocations and frees, Rust only borrows and drops memory

Ownership and lifetimes
Examples

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

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

Ownership and lifetimes
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);
}
```

Guessing game
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!"),
        }
    }
}
```

Guessing game
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;
            }
        }
    }
}
```

Guessing game
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--
```

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
```

Thank you!

Interesting readings

Creators Admit UNIX, C Hoax
C++?? : A Critique of C++ (or Programming and Language Trends of the 1990s)
Mastering the AP CS Curriculum Without Using C++
Why not Rust?
Why is Rust the Most Loved Programming Language?
No, C++ still isn't cutting it.
Falling in love with Rust
Rust after the honeymoon
The relative performance of C and Rust