Rust Intro

Jayon Reis

Disclaimer

I am not a pro in rust and I suck explaining stuff and I don't know much about compilers and stuff. It is my general impression.

History

- Created in 2006 by a Mozilla employee in his spare time.
- In 2009 Mozilla started sponsoring it.
- Original compiler was written in OCaml and it got rewritten in rust and was able to compile itself in 2011 and uses llvm as backend.
- First stable version was released in 2015.
- Mozilla uses it to develop servo, a browser engine.

Hello world

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

```
Hello world!
```

Features

- Fast as C and flexible as C++.
- Memory safe, no leaks and it does not use a GC.
- Abstraction without cost.
- Easy to understand after you fight with the error system.
- It uses pattern matching instead of throwing exceptions or returning a parameter with error and the other with the value.

Things I like

Matches

```
#[derive(Debug)]
enum Foo {
      A = 1,
      B = 2,
      C = 3
}

struct Bar {
    baz: Foo
}
```

```
fn main() {
    let a = Bar { baz: Foo::A };
    let b = Bar { baz: Foo::B };
    let c = Bar { baz: Foo::C };
    match a.baz {
        Foo::A => println!("Match a {:?}", Foo::A),
        _ => {}
    }
    match b.baz {
        Foo::B => println!("Match b {:?}", Foo::B),
        _ => {}
    }
    match c.baz {
        unknown => println!("Unknown match {:?}", unknown)
    }
}
```

```
Match a A
Match b B
Unknown match C
```

Borrow system

What will happen here?

```
#[derive(Debug)]
struct Foo;

fn borrow(a: Foo) {
    println!("Borrowed {:?}", a);
}

fn main() {
    let a = Foo {};
    borrow(a);
    println!("{:?}", a);
}
```

```
error: use of moved value: `a` [--explain E0382]
  --> <anon>:11:22
   |>
10 |>     borrow(a);
   |>            - value moved here
11 |>     println!("{:?}", a);
   |>                      ^ value used here after move
<std macros>:2:27: 2:58: note: in this expansion of
        format_args!
<std macros>:3:1: 3:54: note: in this expansion of print!
        (defined in <std macros>)
<anon>:11:5: 11:25: note: in this expansion of println!
        (defined in <std macros>)
note: move occurs because `a` has type `Foo`, which does
        not implement the `Copy` trait

error: aborting due to previous error
```

Fixing it

```
#[derive(Debug)]
struct Foo;

fn borrow(a: &Foo) {
    println!("Borrowed {:?}", a);
}

fn main() {
    let a = Foo {};
    borrow(&a);
    println!("{:?}", a);
}
```

```
Borrowed Foo
Foo
```

Another fix

```
#[derive(Debug)]
struct Foo;

fn borrow(a: Foo) -> Foo {
    println!("Borrowed {:?}", a);
    a
}

fn main() {
    let mut a = Foo {};
    a = borrow(a);
    println!("{:?}", a);
}
```

Mutability

```
fn main() {
    let a = 1;
    a += 1;
    println!("{}", a);
}
```

```
error[E0384]: re-assignment of immutable variable `a`
 --> <anon>:3:5
  |
2 |     let a = 1;
  |         - first assignment to `a`
3 |     a += 1;
  |     ^^^^^^ re-assignment of immutable variable

error: aborting due to previous error
```

```rust
fn main() {
    let mut a = 1;
    a += 1;
    println!("{}", a);
}
```

```
2
```

Thread safety

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[0] += i;
        });
    }
    thread::sleep(Duration::from_millis(50));
}
```

```
error[E0382]: capture of moved value: `data`
 --> <anon>:9:13
  |
8 |         thread::spawn(move || {
  |                       ------- value moved
  |                               (into closure) here
9 |             data[0] += i;
  |             ^^^^ value captured here after move
  |
  = note: move occurs because `data` has type
          `std::vec::Vec<i32>`, which does not implement
          the `Copy` trait

error: aborting due to previous error
```

Fixing it

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let lock = data.clone();
        thread::spawn(move || {
            let mut data = lock.lock().unwrap();
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
    println!("Output value: {:?}", data);
}
```

Output

```
Output value: Mutex { data: [4, 2, 3] }
```

- You cannot share global writable data between threads without locks.
- You can share readable data between threads without locks.

Modularity

A lot of "stdlib" modules are crates [1] like rand [2] which for me is amazing because packages can react better to changes i.e. security fixes are easier to deploy as an external modules than the whole language.

[1] https://crates.io
[2] https://crates.io/crates/rand

Exceptions

There are none!
You use macros like `let value = try!(function())`
which would translate for something like:

```
let value = match function() {
    Some(value) => value,
    Err(err) => return Err(err)
}
println!("Value is: {:?}", value);
```

on nightly you can activate `?` to make it shorter like `function()?`

Other stuff

- A function that can fail, returns a `Result` which can contain `Ok(value)` or `Err(err)` and instead of returning tuples (like in go) and checking if error is not null, you just use the `try!` macro, `unwrap` or the other functions like `map`, `expect_or` and etc...
- There is no concept of `null`, you have `Option` that can have `Some(value)` or `None` and these values must e unpacked some way before doing some stuff like `function().non_existant_field`, you would have to do at least `function()?.field` and it would break the current function if None was returned.
- Generics and traits, but too big to talk here.

Demo time!

Initialize the project:

- `curl https://sh.rustup.rs -sSf | sh -s` this will install rust up, something like pyenv.
- `rustup update nightly`
- `cargo new --bin example`
- `cd example`
- `rustup override add nightly`
- `cargo run`

Setting up the database

Thank you