In Rust We Trust

肖翊成，朱家润，陈贲

Table of contents

Intro

What is Rust?

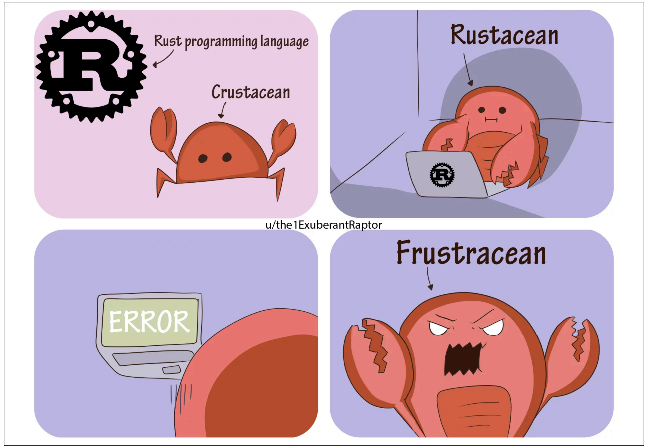
Features:

- 🦀 High Security
- ⚡ High Performance

Kills the Bugs ❎

⇒ Kills the Programmers who write Bugs.

- Default Immutable
- Ownership
- lifetime



Default Immutable

```c
#include<stdio.h>

int main() {
    int a = 5;
    a += 1;
    printf("%d", a);

    return 0;
}
```

```
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:4:18
  |
2 |     let a = String::from("Hello, World! ");
  |         - move occurs because `a` has type `String`, which does not implement the `Copy` trait
3 |     let b = a;
  |             - value moved here
4 |     print!("{}", a);
  |                  ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args` which comes from the expansion of the macro `print`
(in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
```

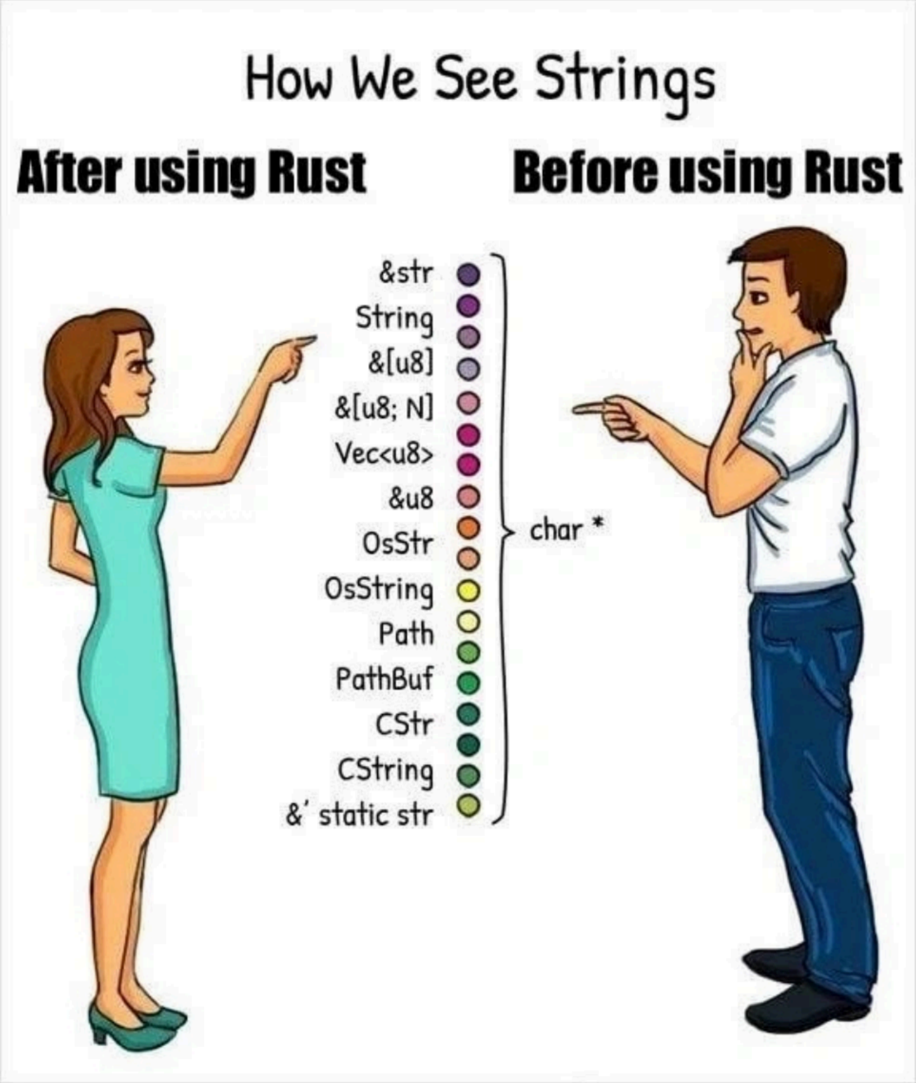
Ownership

```
#include <stdio.h>

int main()
{
   char* a = "Hello, World! ";
   char* b = a;
   printf("%s", a);
   printf("%s", b);

   return 0;
}
```

```
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:4:18
  |
2 |     let a = String::from("Hello, World! ");
  |         - move occurs because `a` has type `String`, which does not implement the `Copy` trait
3 |     let b = a;
  |             - value moved here
4 |     print!("{}", a);
  |                  ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args` which comes from the expansion of the macro `print` (
help: consider cloning the value if the performance cost is acceptable
```


How We See Strings
After using Rust
Before using Rust
&str
String
&[u8]
&[u8; N]
Vec<u8>
&u8
OsStr
OsString
Path
PathBuf
CStr
CString
&' static str
char *

lifetime

Strong compiler ensures security

```
fn main() {
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:13
  |
4 |         let x = 5;
  |             - binding `x` declared here
5 |         r = &x;
  |             ^^ borrowed value does not live long enough
6 |     }
  |     - `x` dropped here while still borrowed
7 |     println!("r: {}", r);
  |                       - borrow later used here

For more information about this error, try `rustc --explain
```

About Rust

`rustc` is what makes Rust great!

Why Rust?

What we talk when we talk about Rust?

When we talk about Rust, we refer to words like **memory safety**, **efficiency**, **expressiveness**, **open-source** and so on… Nowadays, when we talk about system safety, we can't avoid mentioning Rust.

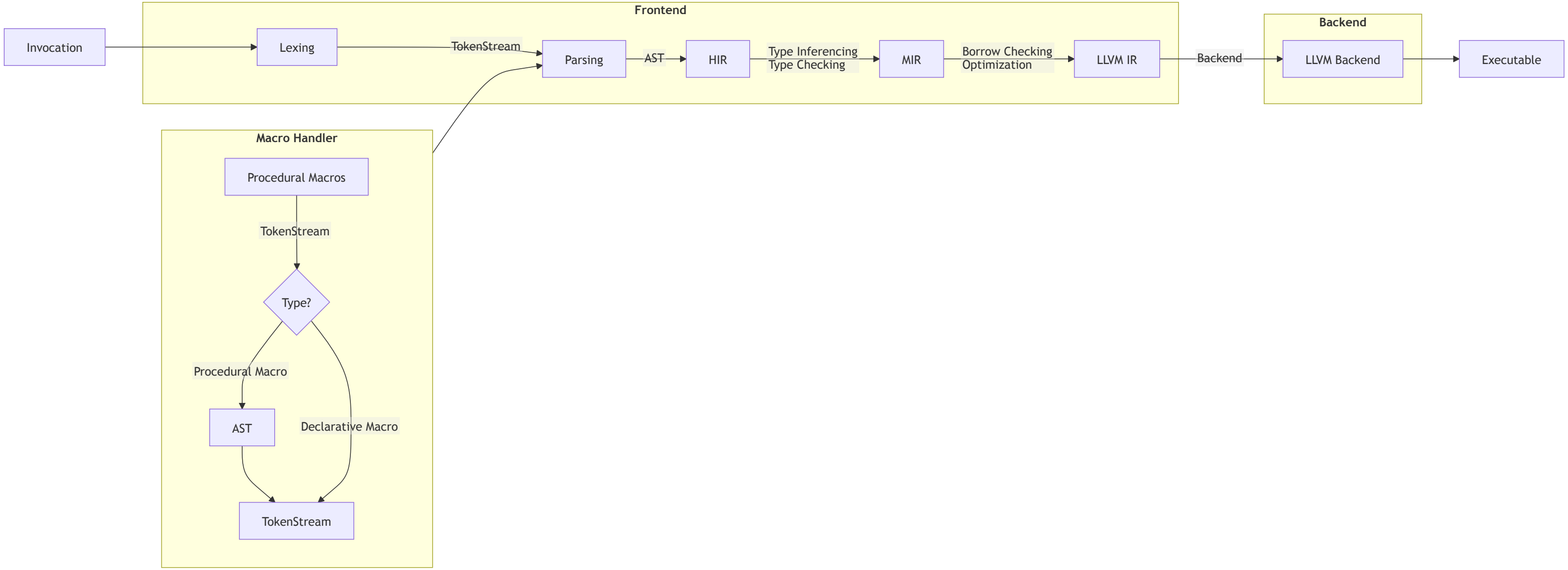
What makes Rust different from other languages?

Rust Compiler `rustc` may give you the answers.

- 🛠️ It has a very powerful frontend system whereas the backend is backed up by the famous LLVM. (The same backend for compilers like clang/intel DPC/C++ compilers)
- 🤔 The explicit definition of ownership enables the compiler to perform static analysis on the code, resolving issues only using frontend.

Compiling Rust

Rust Compiler Flow



Ownership and Lifetime in Rust

Using this, we try to demonstrate how static analysis works.

Preface

Good prgrammers select language, while great language selects programmers.

Ownership and **lifetimes** are core concepts in the Rust language. In fact, these concepts exist not only in Rust but also in C/C++. Nearly all memory safety issues stem from the incorrect use of ownership and lifetimes. Any programming language that does not use garbage collection to manage memory faces these challenges.

However, Rust explicitly defines these concepts at the language level and provides language features that allow users to explicitly control ownership transfer and declare lifetimes. Additionally, the compiler checks for various misuse errors, enhancing the program's memory safety.

I believe that these concepts are still way too abstract to understand. But I also believe most of us encounter problems like `segmentation fault`, `dangling pointer`, `buffer overflow`, etc. in our daily programming. These problems are all related to memory safety.

A Simple Example

Write you a Rust for safe

Ownership means control over a memory region associated with a variable. This region can exist in various memory locations (like heap, stack, or code segment). In high-level languages, accessing these memory regions typically requires associating them with variables, unlike in low-level languages where direct access is possible.

```cpp
#include <iostream>
using namespace std;

int main() {
    int values[3]= { 1,2,3 };
    cout<<values[0]<<","<<values[3]<<endl;
    // Buffer overflow
    return 0;
}
```

How do Rust cope with such problems?

From C++ to Rust

In Rust, the **ownership** concept comes with binding instead of assigning likewise in C++.

```cpp
#include <iostream>

int main()
{
    int a = 1;
    std::cout << &a << std::endl;   /* 0x62fe1c */
    a = 2;
    std::cout << &a << std::endl;   /* 0x62fe1c */
}
 // In this case, the address of `a` remains the same. We are assigning values to `a`.
```

Assignment is the act of writing a value into the memory region associated with a variable, while **binding** is the process of establishing the relationship between a variable and a memory region. In Rust, this also involves transferring ownership of that memory region to the variable.

Ownership Transfer and Borrowing

Transfer or Borrow?

With binding concepts, Rust transfers ownership of a memory region to a variable. This transfer is called **ownership transfer**. Transfer of ownership avoids copying the data, which will be more efficient. If we just want to use the data without transferring ownership, we can borrow the data. This is called **borrowing**.

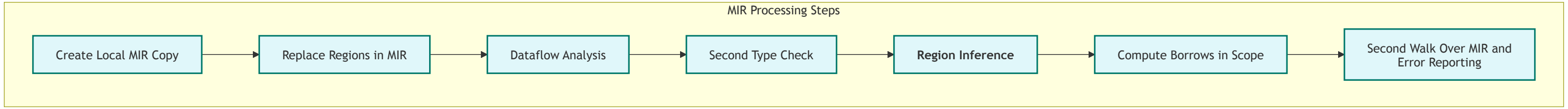
```
fn main() {
    let a = 1;
    let b = a; // Ownership transfer
    println!("a:{}",a); /* Error: use of moved value: `a` */
    println!("b:{}",b); /* 1 */
}
```

In the first example, the ownership of `a` is transferred to `b`. Therefore, `a` cannot be used after the transfer. In the second example, `im_ref` and `mut_ref` are borrowed from `a`. The ownership of `a` is not transferred, so `a` can still be used. However, `mut_ref` cannot be used before `im_ref` is released.
In Rust, the compiler checks for ownership transfer and borrowing. If the code violates these rules, the compiler will throw an error. This is called static analysis.

Borrow Checker in Rust Compilers

ref: Rust Compiler Development Guide

The borrow checker source is located in the `rustc_borrowck` crate. The main entry point is the `mir_borrowck` query.



During `Region Inference` process, the compilers will conduct:

- Constraint Propagation
- Lifetime Parameters Checking
- Member Constraints Checking
- …

Lifetime

How long does the data live? How does Rust compiler analyze it?

The lifecycle of a variable is mainly related to its scope, and in most programming languages, it is implicitly defined. In Rust, however, you can explicitly declare the lifetime parameters of variables, which is a very unique design. This syntactic feature is something that is rarely seen in other languages.

```rust
struct V{v:i32}

fn bad_fn() -> &V{
    let a = V{v:10};
    &a
}

fn main(){
    let res = bad_fn();
}
// [At line 3] error[E0106]: missing lifetime specifier
```

`'a` imposes a requirement on the reference returned within the function body: the data referred to by the returned reference must have a lifetime at least as long as `'a`, meaning it must last at least as long as the variable in the caller context that receives the return value.

More about Lifetime

Lifetime in struct

```
struct G<'a>{ m:&'a str}

fn get_g() -> () {
    let g: G;
    {
        let  s0 = "Hi".to_string();
        let  s1 = s0.as_str();
        g = G{ m: s1 };
    }
    println!("{}", g.m);
}
```

The lifetime definition of a struct ensures that, within an instance of the struct, the lifetime of its reference members is at least as long as the lifetime of the struct instance itself.

Lifetime Checker in Rust Compilers

In a function definition, the compiler does not know what the actual calling context of the function will be in the future. The lifetime parameters essentially serve as a **contract** between the function context and the caller context regarding the lifetimes of the parameters.

Lifetime checking in the caller context

In the caller context, the variable `res` that receives the borrowed value returned by the function cannot have a lifetime longer than the lifetime of the returned borrow (which is derived from the input borrowed parameters). Otherwise, `res` would become a dangling pointer after the input parameters go out of scope.

```
let res: &str;
{
    let s = String::from("reload");
    res = remove_prefix(&s, "re") // s us out of scope
}
println!("{}", res);
```

Thank you!