Why Rust?

Talking about the giant crab in the room.

- Kartavya Vashishtha

🚨 This session is interactive 🚨

- Recommended that you have a laptop with you.
- Install **`cargo`** and **`rust-analyzer`** in your IDE of choice. (also **`gcc`** if you don’t have it)

Resources to use ~~if~~ when you get lost

- Rust in half an hour: https://fasterthanli.me/articles/a-half-hour-to-learn-rust
- The Rust Book: https://doc.rust-lang.org/book/

Reasons to listen to this presentation:

- “**Microsoft rewriting core Windows libraries in Rust**”
  - The Register

- “…**over 1,000 Google developers**… have authored and committed Rust code as some part of their work in 2022”
  - Rust fact vs. fiction: 5 Insights from Google's Rust journey in 2022

- **“**For the past 18 months we have been **adding Rust support to the Android Open Source Project**…”
  - Rust in the Android platform, Google Security Blog

Platinum

Gold

Silver

Rust

A language empowering everyone
to build reliable and efficient software.

GET STARTED

Version 1.73.0

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Put another way:

- Rust is a *systems* language: it makes (most) things explicit.
- Features *borrow checking*, a paradigm to prevent memory unsafety.
- Has a *powerful type system* to shift as many errors to compile time as possible.

Problem 1: Opening a File

The C Approach

```c
#include<stdio.h>

int main () {
    FILE* fp = fopen("example.txt", "r");
    char buf[30] = {0};
    fgets(buf, 30, fp); // safer than gets
    printf("%s", buf);
}
```

The C Approach

Let’s run it.

The C Approach

```
➜  ~/presentations/why-rust ./a.out
[1]    17853 segmentation fault  ./a.out
```

The Rust Approach

```rust
use std::{fs::File, io::Read};

fn main() {
    let f: Result<File, std::io::Error> = File::open("src/main.rs");

    let mut f: File = f.unwrap();

    let mut buf: [u8; 30] = [0; 30];
    f.read_exact(&mut buf).unwrap();
}
```

The Rust Approach (step-by-step)

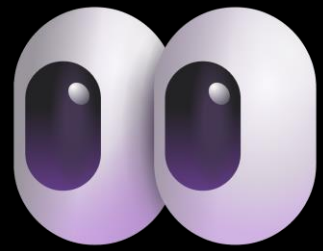
```rust
use std::{fs::File, io::Read};

fn main() {
    let f: Result<File, std::io::Error> = File::open("src/main.rs");
    //...
```

The Rust Approach (step-by-step)^2

```rust
/// `Result` is a type that represents either success ([`Ok`]) or failure ([`Err`]).
///
/// See the [module documentation](self) for details.
#[derive(Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
#[must_use = "this `Result` may be an `Err` variant, which should be handled"]
#[rustc_diagnostic_item = "Result"]
#[stable(feature = "rust1", since = "1.0.0")]
pub enum Result<T, E> {
    /// Contains the success value
    #[lang = "Ok"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Ok(#[stable(feature = "rust1", since = "1.0.0")] T),

    /// Contains the error value
    #[lang = "Err"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Err(#[stable(feature = "rust1", since = "1.0.0")] E),
}
```



The Rust Approach (step-by-step)^2

Enums and structs store data.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Aside: Abstract Data Types

- Sum types can be one of several possibilities (i.e. similar to an enum)
- In mathy language, the set of a sum type is the union of the sets of the constituent types.

- Product types are combinations of several types (i.e. similar to a struct or tuple)
- In mathy language, the set of a product type is the cartesian product of the sets of the constituent types.

The Rust Approach (step-by-step)^2

- Enums and structs store data.

- Can have methods attached to them that may borrow, mutably borrow, or consume their associated objects.

The Rust Approach (step-by-step)^2

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

impl<T, E> Result<T, E> {
    // used as res.unwrap() where res: Result<T, E>
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) => panic!("Error!"),
        }
    }
}
```

The Rust Approach (step-by-step)

```rust
use std::{fs::File, io::Read};

fn main() {
    let f: Result<File, std::io::Error> = File::open("src/main.rs");

    let mut f: File = f.unwrap();
    //...
```

Note that a variable must be declared `mut` to modify it in any way.*

The Rust Approach (step-by-step)

```rust
use std::{fs::File, io::Read};

fn main() {
    //...

    let mut buf: [u8; 30] = [0; 30];
    f.read_exact(&mut buf).unwrap();
}
```

Declare a mutable array of *u8* (unsigned 8-byte integers), and pass a *mutable* reference to `read_exact`.

The Rust Approach (step-by-step)^2

What's the signature of `read_exact`?

```
// somewhere inside the std::io::Read module:
fn read_exact(&mut self, buf: &mut [u8]) -> Result<(), Error> {
    //...
}
```

`buf: &mut [u8]` is an *array slice* that consists of the pointer to the array and its length.

The Rust Approach (step-by-step)

```rust
use std::{fs::File, io::Read};

fn main() {
    let f: Result<File, std::io::Error> = File::open("src/main.rs");
    //...
    let mut buf: [u8; 30] = [0; 30];
}
```

Note that the code doesn’t explicitly close the file anywhere. When f goes out of scope, it runs a *drop* function (destructor) that closes the file automatically.

Take a breath.
We’re just getting started.

Problem 2:
Iterating Over a Vector

Iterating in C++

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    std::vector<int> vec{1, 2, 3, 5, 6};

    // Find the element with value 3.
    auto it = find(vec.begin(), vec.end(), 3);
    if (it != vec.end()) { // check if element was found
        vec.insert(it, 4); // insert 4 after 3
    }

    // print the rest of the elements
    for(; it != vec.end(); it++)
        std::cout<< *it << " ";
}
```

Iterating in C++

```
→  ~/presentations/why-rust ./a.out
-674674232 6 0 49 0 1 2 3 4 5 6
```

Iterating in C++

If you think that's contrived,

CVE-2016-9079 (a vulnerability in Firefox) is caused by code equivalent to:

```
int main()
{
    std::vector<int> vec{1, 2, 3, 5, 6};

    int last = 0;
    for (auto it = vec.begin(); it != vec.end(); it++) {
        if ( is_num_bad(*it) ) {
            vec.insert(it, -1);
        }
    }
}
```

Translating to Rust

```rust
fn main () {
    let mut v: Vec<i32> = vec![1, 2, 3, 4, 5];
    let mut it = v.iter();

    let idx: Option<usize> = it.position(|&x| x == 3);

    if let Some(i) = idx {
        v.insert(i, 4); // insert 4 at index i
    }

    it.for_each(|x| print!("{x} "));
}
```

The Option enum

```rust
//...
let idx: Option<usize> = it.position(|&x| x == 3);

if let Some(i) = idx {
    v.insert(i, 4);
}
//...
```

```rust
enum Option<T> {
    Some(T),
    None
}
```

Translating to Rust

```rust
fn main () {
    let mut v: Vec<i32> = vec![1, 2, 3, 4, 5];
    let mut it = v.iter();

    let idx: Option<usize> = it.position(|&x| x == 3);

    if let Some(i) = idx {
        v.insert(i, 4);
    }

    it.for_each(|x| print!("{x} "));
}
```

Let’s run it!

```
➜  ~/presentations/why-rust cargo build
   Compiling why-rust v0.1.0 (/home/desmond-lin-7/presentations/why-rust)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
  --> src/main.rs:88:13
   |
83 |         let mut it = v.iter();
   |                      -------- immutable borrow occurs here
...
88 |             v.insert(idx, -1);
   |             ^^^^^^^^^^^^^^^^^ mutable borrow occurs here
...
91 |         it.for_each(|x| print!("{x}"));
   |         -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
```

Having a look at the signatures…

And we find:

```
pub fn iter(&self) -> ... { ... }

pub fn insert(&mut self, index: usize, element: T)
```

XOR Mutability

- Can take a reference to a value with `&` or `&mut`.
- Can have either:
  - any number of `&` references live at one time.
  - **or** a single `&mut` reference.

- `&` references are write-only.
- `&mut` references are read and write.

XOR Mutability Example 1:

```
fn main() {
    let s = String::from("hello");
    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
cannot borrow `*some_string` as mutable, as it is behind a `&`
reference`some_string` is a `&` reference, so the data it refers to
cannot be borrowed as mutable
```

XOR Mutability Example 2:

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
```

```
cannot borrow `s` as mutable more than once at a time
```

But what if I know what I’m doing?

Rust does have an escape hatch to disable XOR mutability.

It’s called `unsafe`.

There’s an entire second book (The Rustonomicon) for that.

Unsafe is very hard to write correctly, but the situation is improving.

Problem 3:
Returning a Pointer

The C Approach

```c
struct example {int a; int b;};

int* get_a(struct example *e) {
    return &e->a;
}

int main () {
    struct example *e = malloc(sizeof(struct example));
    e->a = 1; e->b = 2;
    int* a = get_a(e);

    free(e);

    printf("%d\n", *a);
}
```

The C Approach

```
→  ~/presentations/why-rust ./a.out
1666543595
```

Translating to Rust

```
struct Example {a: i32, b: i32}

fn get_a (e: &Example) -> &i32 {
    &e.a
}

fn main () {
    let ex = Example { a: 1, b: 2 };
    let a = get_a(&ex);

    drop(ex);

    print!("{}", a);
}
```

Let’s run it!

```
➜  ~/presentations/why-rust cargo build
   Compiling why-rust v0.1.0 (/home/desmond-lin-7/presentations/why-rust)
error[E0505]: cannot move out of `ex` because it is borrowed
   --> src/lib.rs:100:10
    |
98  |     let a = get_a(&ex);
    |                   --- borrow of `ex` occurs here
99  |
100 |     drop(ex);
    |          ^^ move out of `ex` occurs here
101 |
102 |     print!("{}", a);
    |                  - borrow later used here
```

Let’s run it!

```
➜  ~/presentations/why-rust cargo build
   Compiling why-rust v0.1.0 (/home/desmond-lin-7/presentations/why-rust)
error[E0505]: cannot move out of `ex` because it is borrowed
   --> src/lib.rs:100:10
    |
98  |     let a = get_a(&ex);
    |                   --- borrow of `ex` occurs here
99  |
100 |     drop(ex);
    |          ^^ move out of `ex` occurs here
101 |
102 |     print!("{}", a);
    |                  - borrow later used here
```

```
pub fn drop<T>(x: T) {}
```

Move Semantics

Seek shelter.
Wall of text incoming.

Move Semantics

Because variables are in charge of freeing their own resources, resources can only have one owner. This also prevents resources from being freed more than once. Note that not all variables own resources (e.g. references).

When doing assignments (`let x = y`) or passing function arguments by value (`foo(x)`), the ownership of the resources is transferred. In Rust-speak, this is known as a move.

After moving resources, the previous owner can no longer be used. This avoids creating dangling pointers.

Type Shenanigans

Lambdas

Rust takes a lot of influence from functional languages.

It features:

- Lambdas / anonymous functions / Closures

```
let add_one = |x| x + 1;
let add_one = |x: i32| x + 1;
```

Traits

Rust also has interfaces (traits)

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Any type that wishes to implement Iterator must prove a next method, and a type for the return item.

Traits

Example implementation:

```
impl Iterator for MyStruct {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        todo!("return the next item")
    }
}
```

Traits

Can also have default implementations:

```
pub fn iterator_stuff(v: Vec<u32>) -> Vec<u32> {
    v.into_iter() // get an iterator from the vector
        .filter(|&x| x % 2 == 0)
        .map(|x| x * 2)
        .collect()
}
```

Implementing Iterator's next gives you access to ~76 methods (fold, flatten, reduce, map, filter, etc.)

Impls can be conditional

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
impl<T, E: Debug> Result<T, E> {
    fn expect(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) => panic!("Error: {:?}", err),
        }
    }
}
```

We implement `expect` for all `Result<T, E>` where `E: Debug`

Bonus: The Typestate Pattern

```
struct Start; // expecting status line
struct Headers; // expecting headers or body

trait ResponseState {}
impl ResponseState for Start {}
impl ResponseState for Headers {}

struct HttpResponse<S: ResponseState> {
    status_code: u16,
    body: Vec<u8>,
    _marker: std::marker::PhantomData<S>, // have to use S somewhere
}
```

Bonus: The Typestate Pattern

```rust
fn new_response() -> HttpResponse<Start> { todo!() }

impl HttpResponse<Start> {
    fn set_status_code(self, status_code: String) -> HttpResponse<Headers> {
      todo!()
    }
}

impl HttpResponse<Headers> {
    fn set_body(self, body: Vec<u8>) -> HttpResponse<Headers> { todo!() }
}
```

Bonus: Explicit Lifetimes

```
pub fn find_line_starts_with<'a>(needle: &str, haystack: &'a str)
    -> Vec<&'a str>
{
    let mut headings = Vec::new();
    for line in haystack.lines() {
        if line.starts_with(needle) {
            headings.push(line);
        }
    }
    headings
}
```

Lifetime of returned vector depends on lifetime of haystack (`'a`)

Project:
Running Average Temperature

```
git clone

https://github.com/DesmondWillowbrook/
rust-presentation-project
```

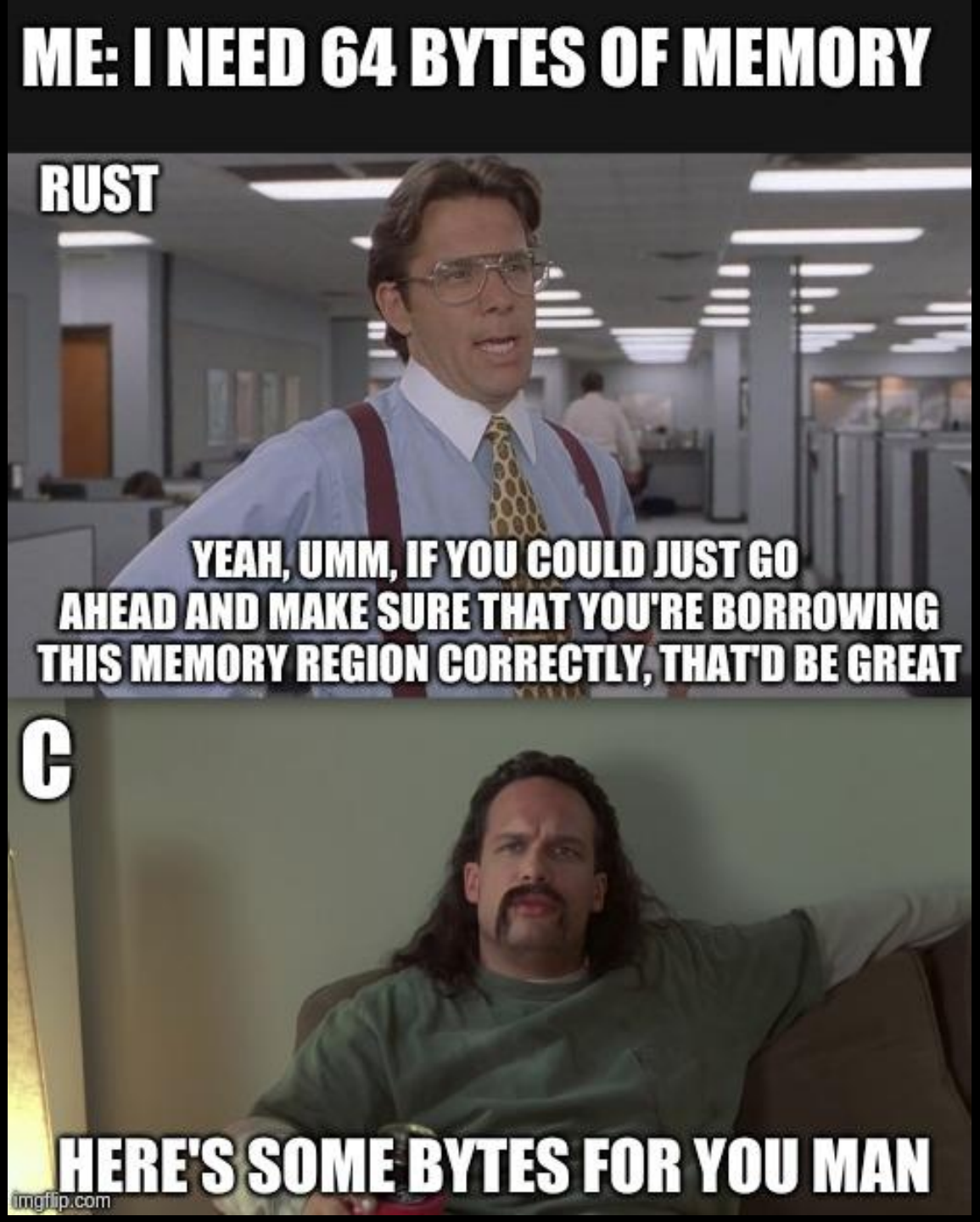
This is where the slides end.

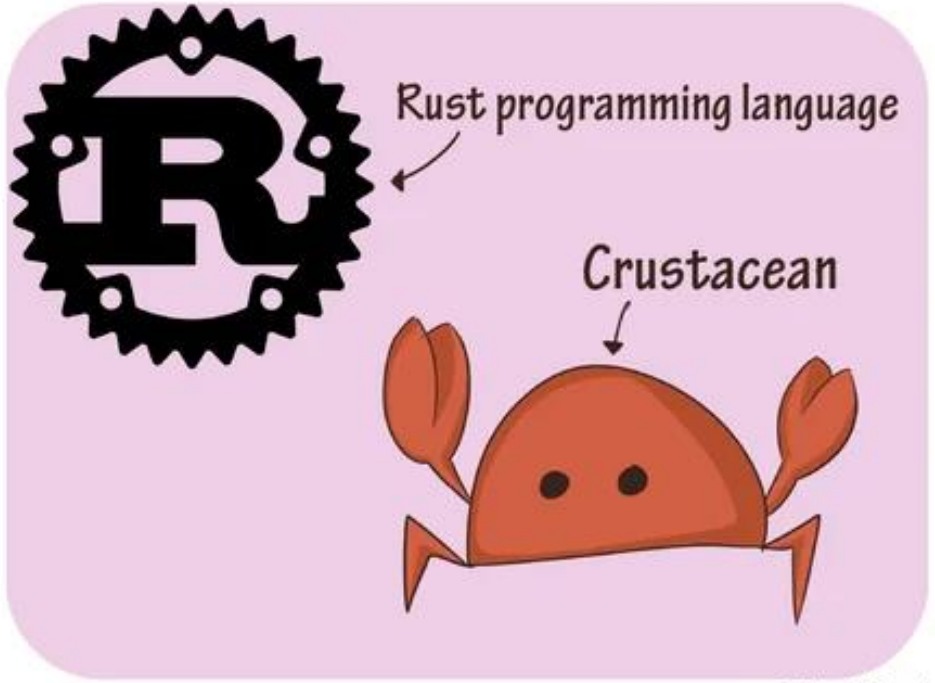
How about we take this to the whiteboard?

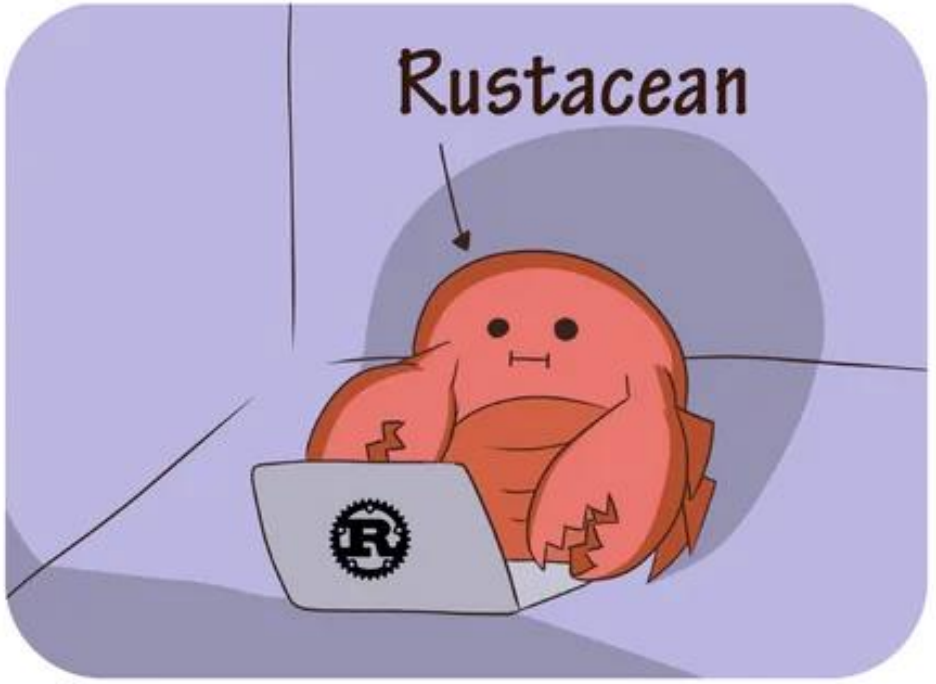
A few memes to calm down


The Rust
Programming
Language
REWRITE
IT IN RUST

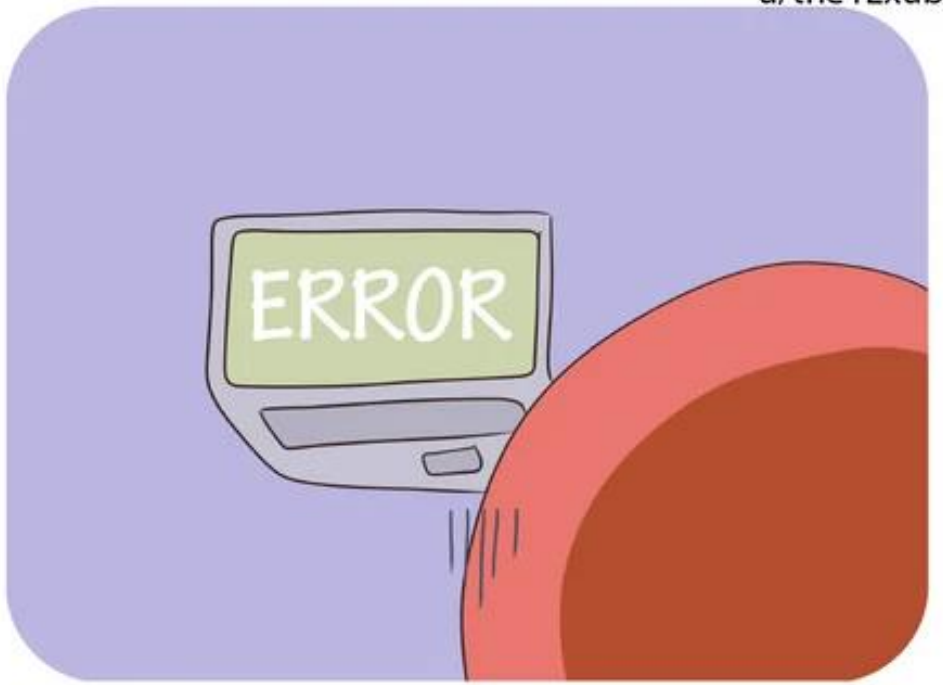
Me: my laptop is a bit slow, hmm hey `top` what's going on
Top: *without even looking up, points at rustc*
Me: ... how much cpu are you using rustc
Rustc: *cores falling out of mouth* *mumbles* 763%


ME: I NEED 64 BYTES OF MEMORY
RUST
YEAH, UMM, IF YOU COULD JUST GO AHEAD AND MAKE SURE THAT YOU'RE BORROWING THIS MEMORY REGION CORRECTLY, THAT'D BE GREAT
C
HERE'S SOME BYTES FOR YOU MAN
imgflip.com





u/the1ExuberantRaptor



Improving the unwrap function

```
enum Result<I, E> {
    Ok(I),
    Err(E),
}
impl<I, E> Result<I, E> {
    fn unwrap(self) -> I {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) => panic!("Error!"),
        }
    }
}
```

We want to print the error.

Improving the unwrap function

```
enum Result<I, E> {
    Ok(I),
    Err(E),
}
impl<I, E> Result<I, E> {
    fn unwrap(self) -> I {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) => panic!("Error!"),
        }
    }
}
```

We want to print the error.
But not all types can be printed.

Enter: Traits

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```

`Formatter` is type that you can write text to.

Enter: Traits

```
struct Position {
    longitude: f32,
    latitude: f32,
}

impl Debug for Position {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "({}, {})", self.longitude, self.latitude)
    }
}
```

Improving the unwrap function

```
enum Result<I, E> {
    Ok(I),
    Err(E),
}
impl<I, E: Debug> Result<I, E> {
    fn expect(self) -> I {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) => panic!("Error: {:?}", err),
        }
    }
}
```

{:?} calls the Debug implementation on type E

Post-talk notes

- Lifetime confusion: slide “Explicit lifetimes” returned lifetimes expire when source lifetime expires because of mutable reference created to buffer.
- Talk about ownership model more explicitly. Probably omit self, &self, &mut self in favor of explicit parameter passing.