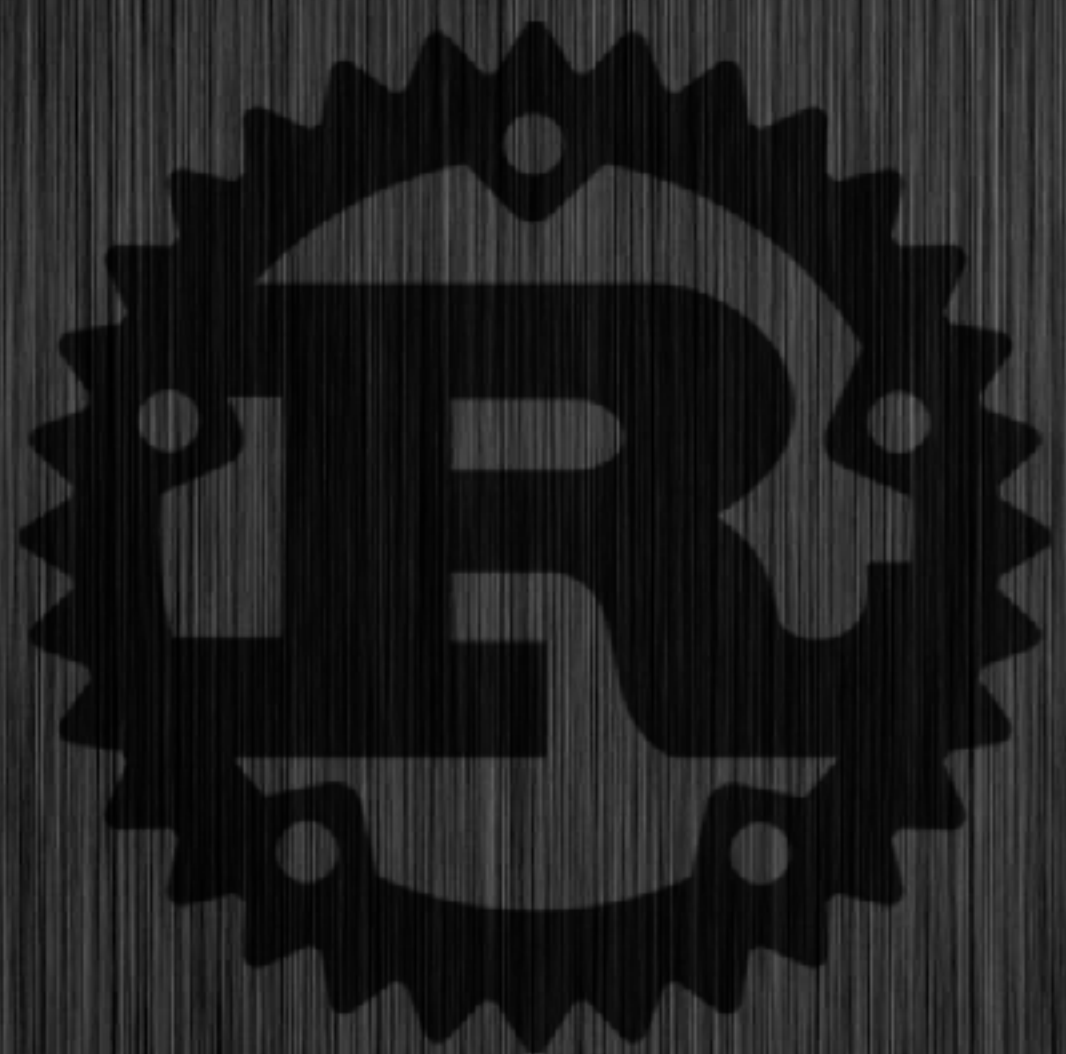


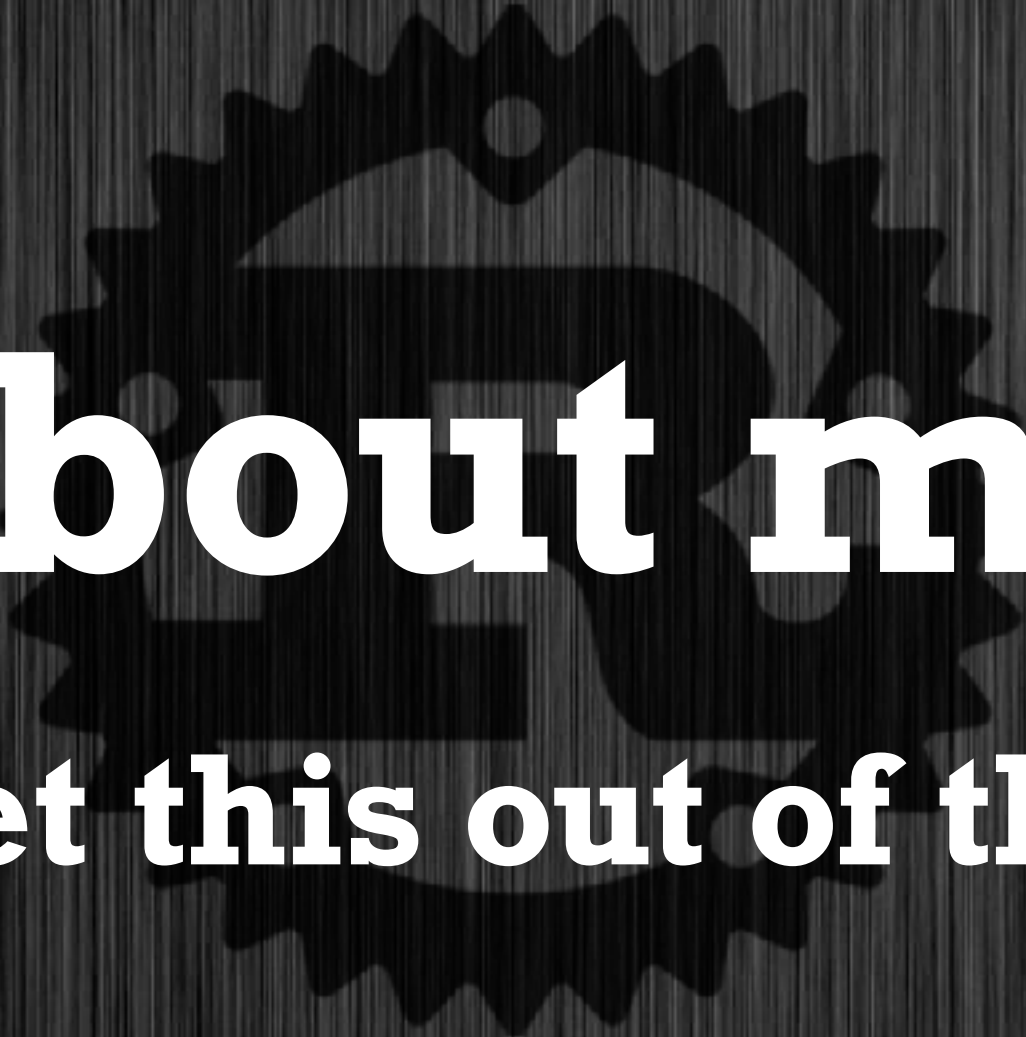


before we start



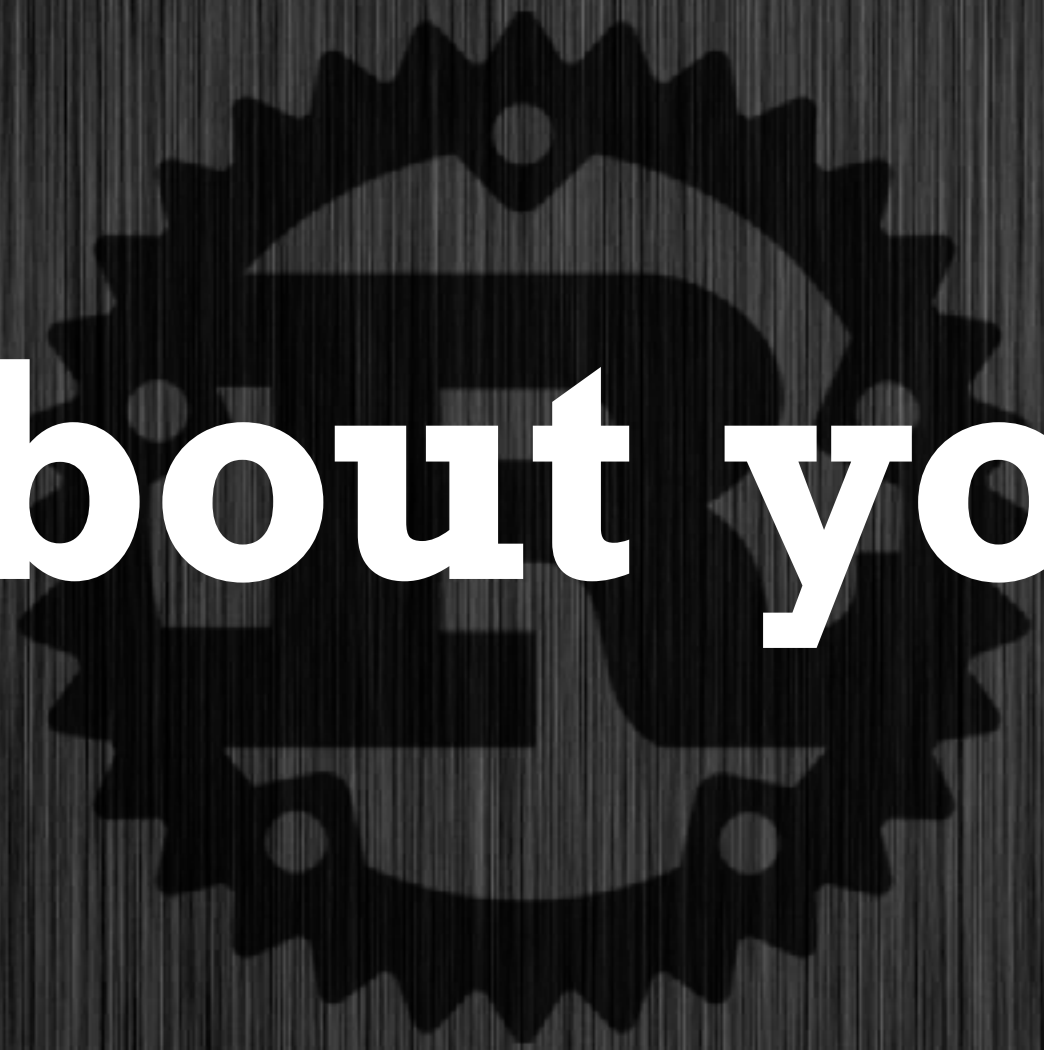
NobleProg



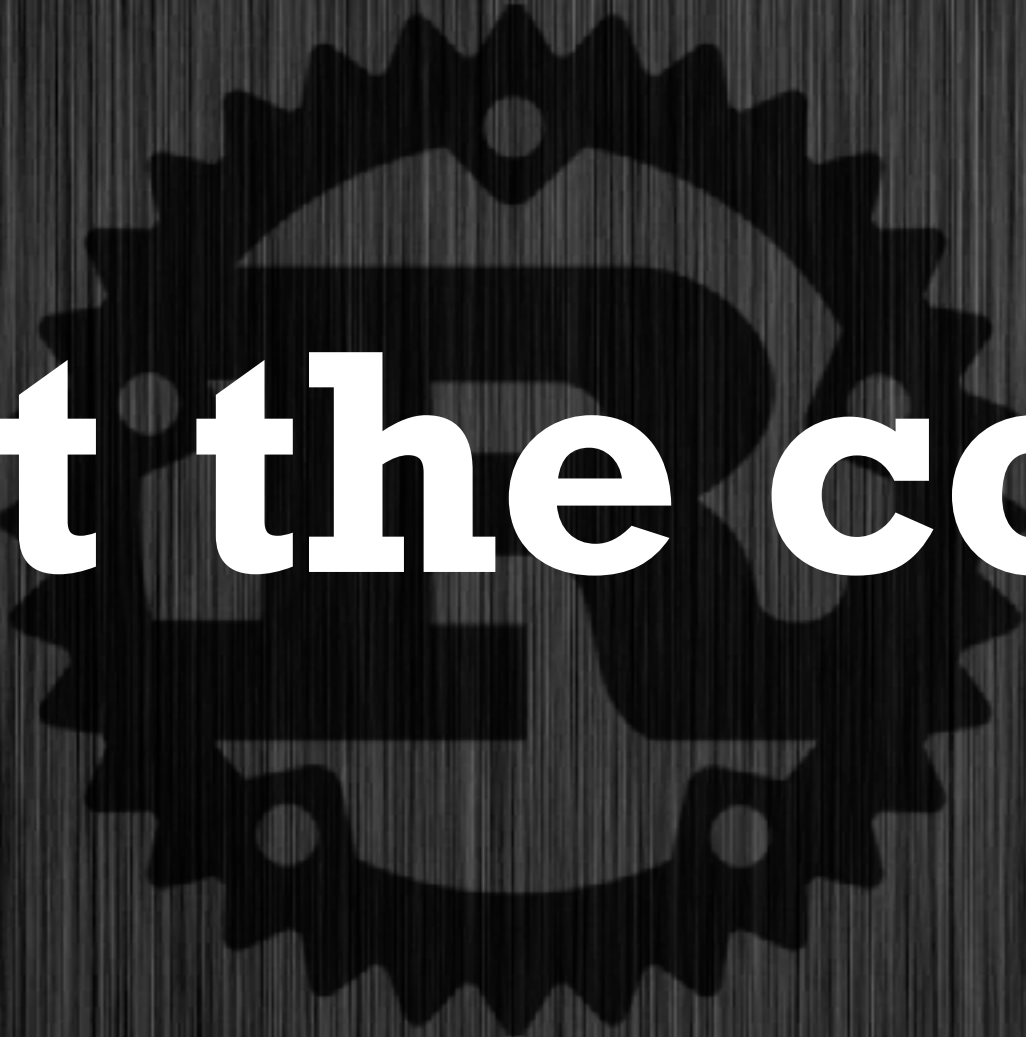


about me

let's get this out of the way



about you

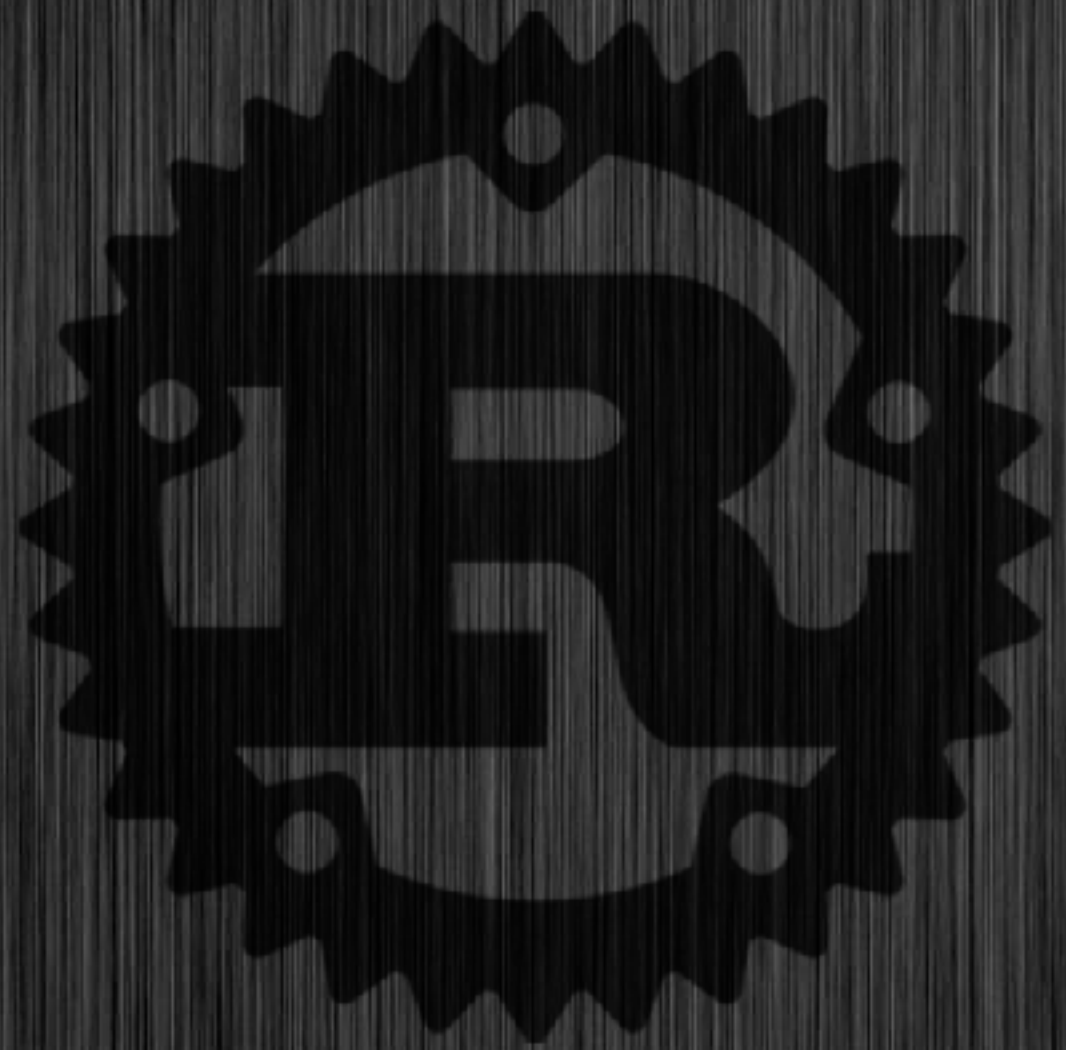


about the course

rust.lab.localdomain.pl



introduction to Rust



introduction to Rust

coding exercises

**write an HTTP cache warmer
in 11 easy steps**

introduction to Rust



digging deeper

**testing
unsafe
FFI**

introduction to Rust

coding exercises

digging deeper

web apps

**write an HTTP cache warmer
in 11 easy steps**

**testing
unsafe
FFI**

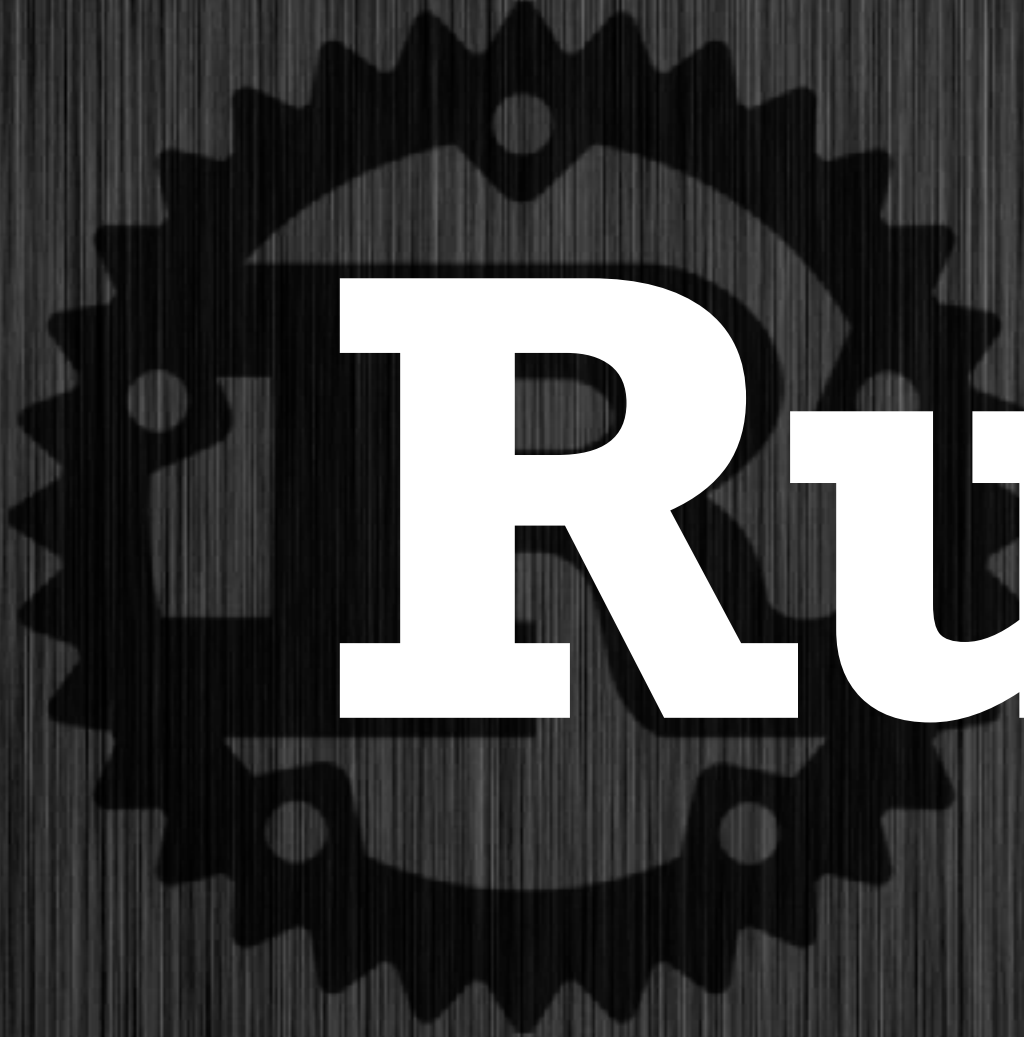
**writing web
applications
with Actix**



programming

in

Rust



Grzegorz Nosek



why Rust?

```
#include <cassert>
#include <vector>

int main()
{
    std::vector<char> vec(10, 'C');
    char* elt = &vec[5];
    vec.resize(5000);

    *elt = '+';
    assert(vec[5] == '+');
}
```

1. Create a vector

2. Get a reference to an element

3. Force a reallocation

4. Use previous reference

```
#include <cassert>
#include <vector>

int main()
{
    std::vector<char> vec(10, 'C');
    char* elt = &vec[5];
    vec.resize(5000);

    *elt = '+';
    assert(vec[5] == '+'); // oops
}
```

1. Create a vector

2. Get a reference to an element

3. Force a reallocation

4. Use previous reference

5. ... that's now invalid

```
fn main()  
{  
    let mut vec = vec!['R'; 10];  
  
    let elt = &vec[5];  
    vec.resize(5000, 's');  
  
    println!("{}", elt);  
}
```

1. Create a vector

2. Get a reference to an element

3. Force a reallocation

4. Use previous reference

... not so fast

```
fn main()  
{  
    let mut vec = vec!['R'; 10];  
  
    let elt = &vec[5];  
    vec.resize(5000, 's');
```

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable  
--> examples/vec.rs:6:2
```

```
5 | |     let elt = &vec[5];  
  | |         --- immutable borrow occurs here  
6 | |     vec.resize(5000, 's');  
  | |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here  
7 | |  
8 | |     println!("{}", elt);  
  | |         --- immutable borrow later used here
```

```
#include <cassert>
#include <thread>
#include <vector>

constexpr const int LOOPS = 100000000;
constexpr const int NTHREADS = 32;

void spin(int* counter)
{
    for (int i = 0; i < LOOPS; ++i)
        ++*counter;
}

int main()
{
    int counter = 0;
    std::vector<std::thread> threads;

    for (int i = 0; i < NTHREADS; ++i)
        threads.emplace_back(spin, &counter);

    for (auto& thread: threads)
        thread.join();

    assert(counter == NTHREADS * LOOPS); // oops
}
```

1. Spawn some threads

2. ... that all write the same variable...

3. ...without locking

```
#include <cassert>
#include <thread>
#include <vector>

constexpr const int LOOPS = 100000000;
constexpr const int NTHREADS = 32;

void spin(int* counter)
{
    for (int i = 0; i < LOOPS; ++i)
        ++*counter;
}

int main()
{
    int counter = 0;
    std::vector<std::thread> threads;

    for (int i = 0; i < NTHREADS; ++i)
        threads.emplace_back(spin, &counter);

    for (auto& thread: threads)
        thread.join();

    assert(counter == NTHREADS * LOOPS); // oops
}
```



**load from *counter
increment register
store to *counter**

```
#include <cassert>
#include <thread>
#include <vector>

constexpr const int LOOPS = 100000000;
constexpr const int NTHREADS = 32;

void spin(int* counter)
{
    for (int i = 0; i < LOOPS; ++i)
        ++*counter;
}

int main()
{
    int counter = 0;
    std::vector<std::thread> threads;

    for (int i = 0; i < NTHREADS; ++i)
        threads.emplace_back(spin, &counter);

    for (auto& thread: threads)
        thread.join();

    assert(counter == NTHREADS * LOOPS); // oops
}
```



load from *counter
increment register
load from *counter
increment register
store to *counter
store to *counter

```
#include <cassert>
#include <thread>
#include <vector>

constexpr const int LOOPS = 100000000;
constexpr const int NTHREADS = 32;

void spin(int* counter)
{
    for (int i = 0; i < LOOPS; ++i)
        ++*counter;
}

int main()
{
    int counter = 0;
    std::vector<std::thread> threads;

    for (int i = 0; i < NTHREADS; ++i)
        threads.emplace_back(spin, &counter);

    for (auto& thread: threads)
        thread.join();

    assert(counter == NTHREADS * LOOPS); // oops
}
```

gcc -O2
optimizes the bug away

mostly

load from *counter
add LOOPS
store to *counter

same bug, LOOPS times
harder to hit

```
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: &mut i32)
{
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let mut counter = 0;
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        threads.push(thread::spawn(|| spin(&mut counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

    assert_eq!(counter, LOOPS * NTHREADS);
    Ok(())
}
```



```
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;
```

```
fn spin(counter: &mut i32)
{
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}
```

cannot mutably borrow `counter` multiple times

```
error[E0499]: cannot borrow `counter` as mutable more than once at a time
--> examples/threads3.rs:19:30
```

```
19 |         threads.push(thread::spawn(|| spin(&mut counter)));
    |                                ^^^^^^^^^^^^^^^^^^^^^^^^^
    |                                |           |
    |                                |           | borrows occur due to use of `counter` in closure
    |                                |           | mutable borrow starts here in previous iteration of loop
    |                                |           | argument requires that `counter` is borrowed for `static`
```

```
for thread in threads.into_iter() {
    thread.join().map_err(|_| "failed to join thread")?;
}
```

```
assert_eq!(counter, LOOPS * NTHREADS);
Ok(( ))
```

```
}
```


threads run concurrently with the main thread

```
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: &mut i32)
{
    for _ in 0..LOOPS {
        *counter += 1;
    }
}
```

```
error[E0502]: cannot borrow `counter` as immutable because it is also borrowed as mutable
--> examples/threads3.rs:26:2
```

```
19 |         threads.push(thread::spawn(|| spin(&mut counter)));
    |                                     ^^^^^^^^^^^^^^^^^^^^^
```

```
    |                                     |           |
    |                                     |           | first borrow occurs due to use of `counter` in closure
    |                                     | mutable borrow occurs here
    |                                     argument requires that `counter` is borrowed for `'static`
```

```
...
26 |     assert_eq!(counter, LOOPS * NTHREADS);
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ immutable borrow occurs here
```

```
for thread in threads.into_iter() {
    thread.join().map_err(|_| "failed to join thread")?;
}
```

```
assert_eq!(counter, LOOPS * NTHREADS);
Ok(( ))
```

}

```
use std::thread;

const LOOPS: i32 = 10000000;
const NTHREADS: i32 = 32;

fn spin(counter: &mut i32)
{
    for _ in 0 .. LOOPS {
        *counter += 1;
    }
}

fn main() -> Result<(), &'static str>
{
    let mut counter = 0;
    let mut threads = Vec::new();

    for _ in 0 .. NTHREADS {
        threads.push(thread::spawn(|| spin(&mut counter)));
    }

    for thread in threads.into_iter() {
        thread.join().map_err(|_| "failed to join thread")?;
    }

    assert_eq!(counter, LOOPS * NTHREADS);
    Ok(())
}
```



**don't worry,
we'll fix it later**

**it's hard to write
safe C++**

(but possible)

**it's hard to write
unsafe Rust
(but possible)**

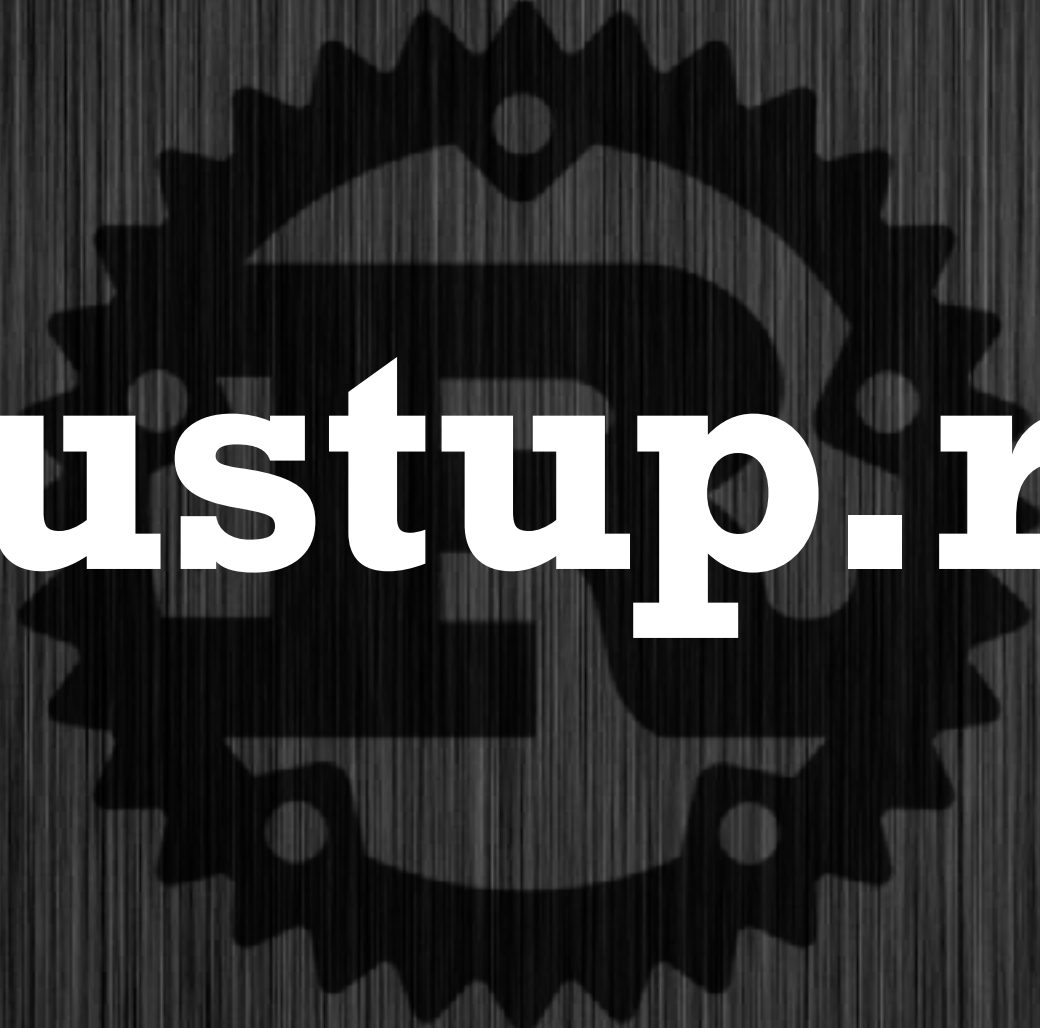
A faint, dark watermark of the Rust logo is centered in the background. It features a stylized 'R' with a gear-like border.

installing Rust

stable | beta | nightly



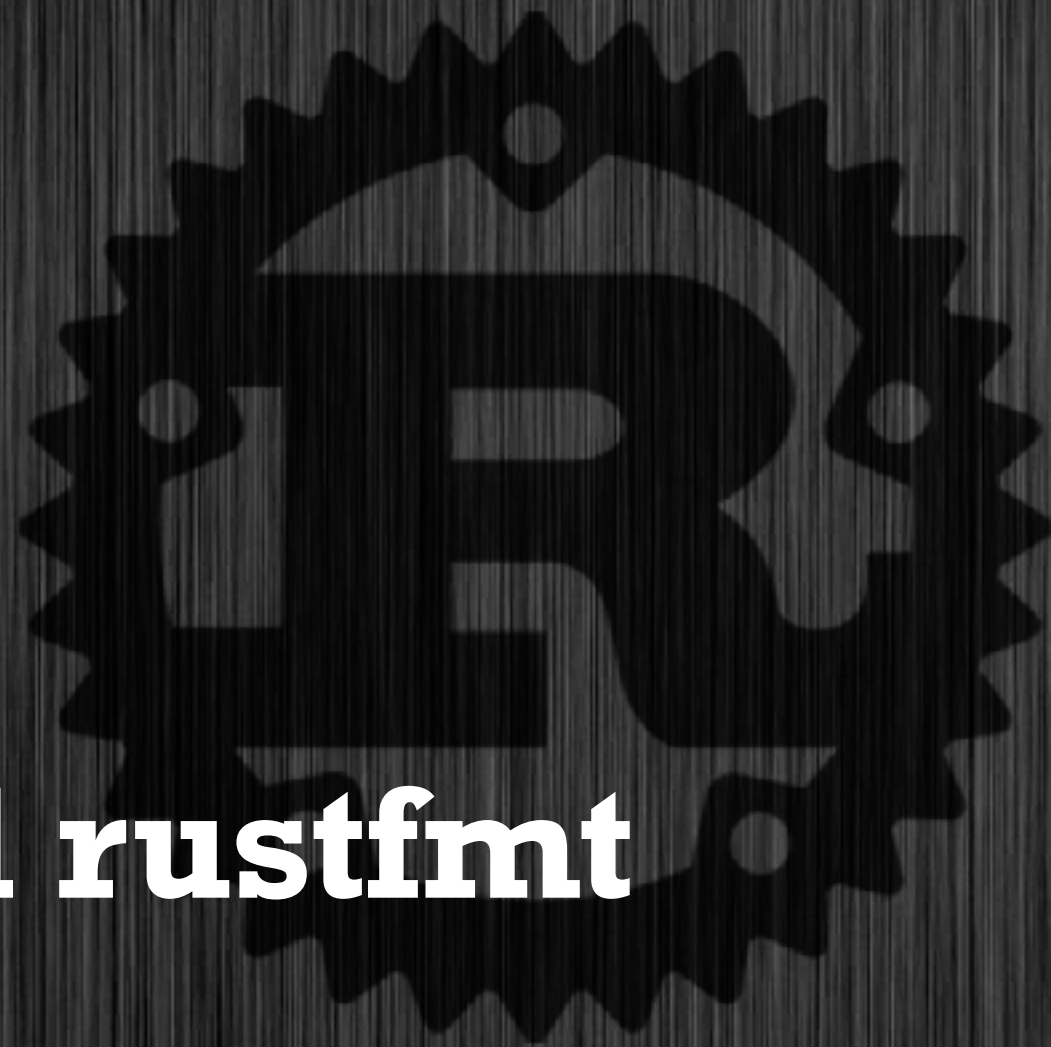
installing Rust



rustup.rs

Hello, cargo!

- **cargo init**
build
run
test
doc
- **rustup component add rustfmt**
cargo fmt
- **rustup component add clippy**
cargo clippy



Hello, world!

```
$ cargo init hello
   Created binary (application) package
$ cd hello/
$ cargo run
   Compiling hello v0.1.0 (/rust/hello)
   Finished dev [unoptimized + debuginfo] target(s) in 0.19s
   Running `target/debug/hello`
Hello, world!
```

what just happened?

cargo init: your first crate

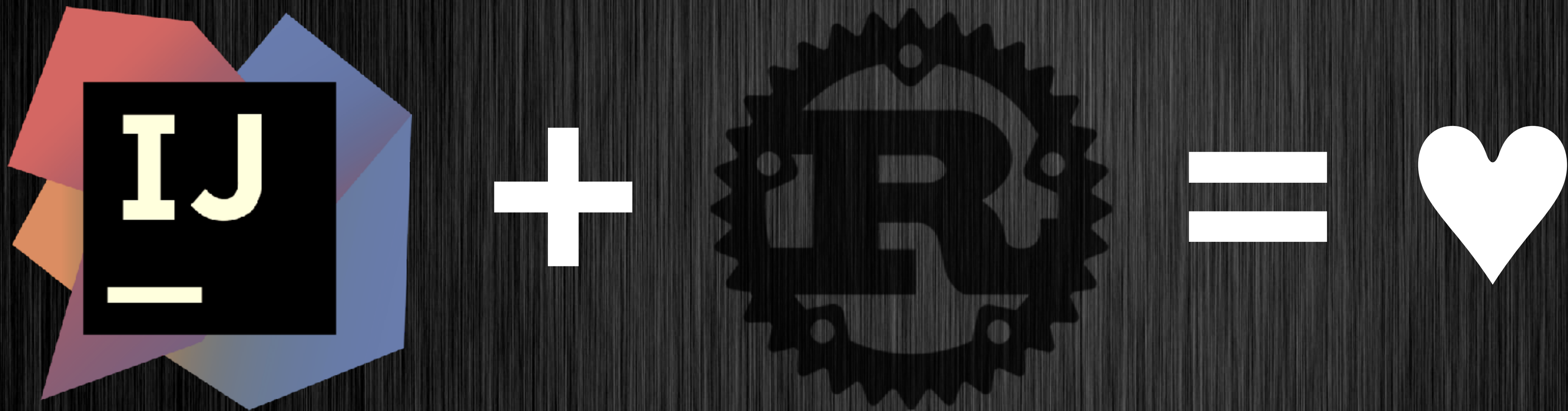
```
.
|-- Cargo.toml
`-- src
    |-- main.rs
```

what just happened?

cargo run



```
.
|-- Cargo.lock
|-- Cargo.toml
|-- src
|   |-- main.rs
|-- target
|   |-- debug
|   :
|   :
|   |-- hello
|   :
```



<https://www.jetbrains.com/idea/>

programming in Rust



syntax basics

syntax basics

```
unsigned long fib(unsigned int n)
{
    if (n < 2)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

```
fn fib(n: u32) -> u64 {
    if n < 2 {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

LL(k) grammar, no Most Vexing Parse

`return` is optional for last expression

```
unsigned long fib(unsigned int n)
{
    if (n < 2)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

```
fn fib(n: u32) -> u64 {
    if n < 2 {
        1
    } else {
        fib(n-1) + fib(n-2)
    }
}
```

`let` introduces a new binding

```
unsigned long fib(unsigned int n)
{
    unsigned long fib_n;

    if (n < 2)
        fib_n = 1;
    else
        fib_n = fib(n-1) + fib(n-2);

    printf("fib(%u) = %lu\n", n, fib_n);
    return fib_n;
}
```

```
fn fib(n: u32) -> u64 {
    let fib_n = if n < 2 {
        1
    } else {
        fib(n-1) + fib(n-2)
    };

    println!("fib({}) = {}", n, fib_n);
    fib_n
}
```

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers = Vec::new();

    for s in num_strings {
        numbers.push(s.parse())
    }

    numbers
}
```

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers = Vec::new();

    for s in num_strings {
        numbers.push(s.parse())
    }

    numbers
}
```

return type is always* explicit

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers = Vec::new();
    for s in num_strings {
        numbers.push(s.parse())
    }

    numbers
}
```

**`numbers` is a vector, but of what?
(this doesn't work in C++)**

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers = Vec::new();

    for s in num_strings {
        numbers.push(s.parse())
    }

    numbers
}
```

since we return it, it must be a vector of
Result<i32, ParseIntError>

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers = Vec::new();

    for s in num_strings {
        numbers.push(s.parse())
    }

    numbers
}
```

**str::parse has a generic return type (also not possible in C++)
but only the implementation for `i32` returns a value of the type
that numbers.push() accepts**

type inference

```
fn parse_numbers(num_strings: &[&str]) -> Vec<Result<i32, ParseIntError>>
{
    let mut numbers: Vec<Result<i32, ParseIntError>> = Vec::<Result<i32, ParseIntError>>::new( );

    for s in num_strings {
        numbers.push(s.parse::

"turbofish"


```

bindings are immutable by default

```
unsigned long fib(unsigned int n)
{
    unsigned long fib_0 = 1;
    unsigned long fib_1 = 1;
    unsigned long fib_n = 0;

    for (unsigned int i = 2; i <= n; ++i)
    {
        fib_n = fib_0 + fib_1;
        fib_0 = fib_1;
        fib_1 = fib_n;
    }

    return fib_n;
}
```

```
fn fib(n: u32) -> u64 {
    let mut fib_0 = 1;
    let mut fib_1 = 1;
    let mut fib_n = 0;

    for _ in 2..=n {
        fib_n = fib_0 + fib_1;
        fib_0 = fib_1;
        fib_1 = fib_n;
    }

    fib_n
}
```

bindings are immutable by default

```
let path = {  
  let mut path = PathBuf::from("/etc");  
  path.push("passwd");  
  path  
};
```

`for` can loop over any Iterator

```
#include <vector>
#include <iostream>
```

```
int main()
{
    std::vector<int> vec = {1, 2, 3, 4, 5, 6};
    for(auto i: vec) {
        std::cout << i << '\n';
    }
}
```

```
fn main() {
    let vec = vec![1, 2, 3, 4, 5, 6];
    for i in &vec {
        println!("{}", i);
    }
}
```

a..b is std::ops::Range { start: a, end: b }

`match` is `switch` on steroids

```
unsigned long fib(unsigned int n)
{
    switch(n) {
        case 0:
        case 1:
            return 1;
        default:
            return fib(n-1) + fib(n-2);
    }
}
```

```
fn fib(n: u32) -> u64 {
    match n {
        0 | 1 => 1,
        other => fib(other-1) + fib(other-2),
    }
}
```

`match` is `switch` on steroids

```
void fizzbuzz(unsigned int n)
{
    for (unsigned int i = 0; i < n; ++i)
    {
        unsigned int mod_3 = (i+1) % 3;
        unsigned int mod_5 = (i+1) % 5;
        if (mod_3 == 0 && mod_5 == 0)
            printf("FizzBuzz\n");
        else if (mod_3 == 0)
            printf("Fizz\n");
        else if (mod_5 == 0)
            printf("Buzz\n");
        else
            printf("%u\n", i+1);
    }
}
```

```
fn fizzbuzz(n: usize) {
    for i in 0..n {
        match ((i+1) % 3, (i+1) % 5) {
            (0, 0) => println!("FizzBuzz"),
            (0, _) => println!("Fizz"),
            (_, 0) => println!("Buzz"),
            (_, _) => println!("{}", i+1),
        }
    }
}
```

`if`, `match` and `loop` are expressions

```
let n = if n % 2 == 0 {  
    n / 2  
} else {  
    3 * n + 1  
};
```

```
let n = match n % 2 {  
    0 => n / 2,  
    _ => 3 * n + 1,  
};
```

```
let prime = loop {  
    let candidate = gen_random();  
    if is_prime(candidate) {  
        break candidate;  
    }  
}
```

memory model: ownership

- **every object is owned by a scope (e.g. a function) and is destroyed when the scope ends**
- **an object can be either moved or borrowed to another scope**
- **once an object is moved, it cannot be accessed again**
- **"fighting the borrow checker"**
fixing bugs before they happen

pass by move

```
fn print(s: String)
{
    println!("{}", s);
}

fn main()
{
    let s = String::from("Hello, world");
    print(s);
    print(s);
}
```

error[E0382]: use of moved value: `s`

--> examples/move.rs:10:8

```
9  |     print(s);
   |           - value moved here
10 |     print(s);
   |           ^ value used here after move
```

= note: move occurs because `s` has type `std::string::String`, which does not implement the `Copy` trait

pass by reference | std::shared_ptr<const T>

```
fn print(s: &String)
{
    println!("{}", s);
}

fn main()
{
    let s = String::from("Hello, world");
    print(&s);
    print(&s);
}
```



```
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/examples/move`
Hello, world
Hello, world
```

pass by mutable reference | T&

```
fn exclaim(s: &mut String)
{
    s.push('!');
}

fn main()
{
    let mut s = String::from("Hello, world");
    exclaim(&mut s);
    exclaim(&mut s);
    println!("{}", s);
}
```

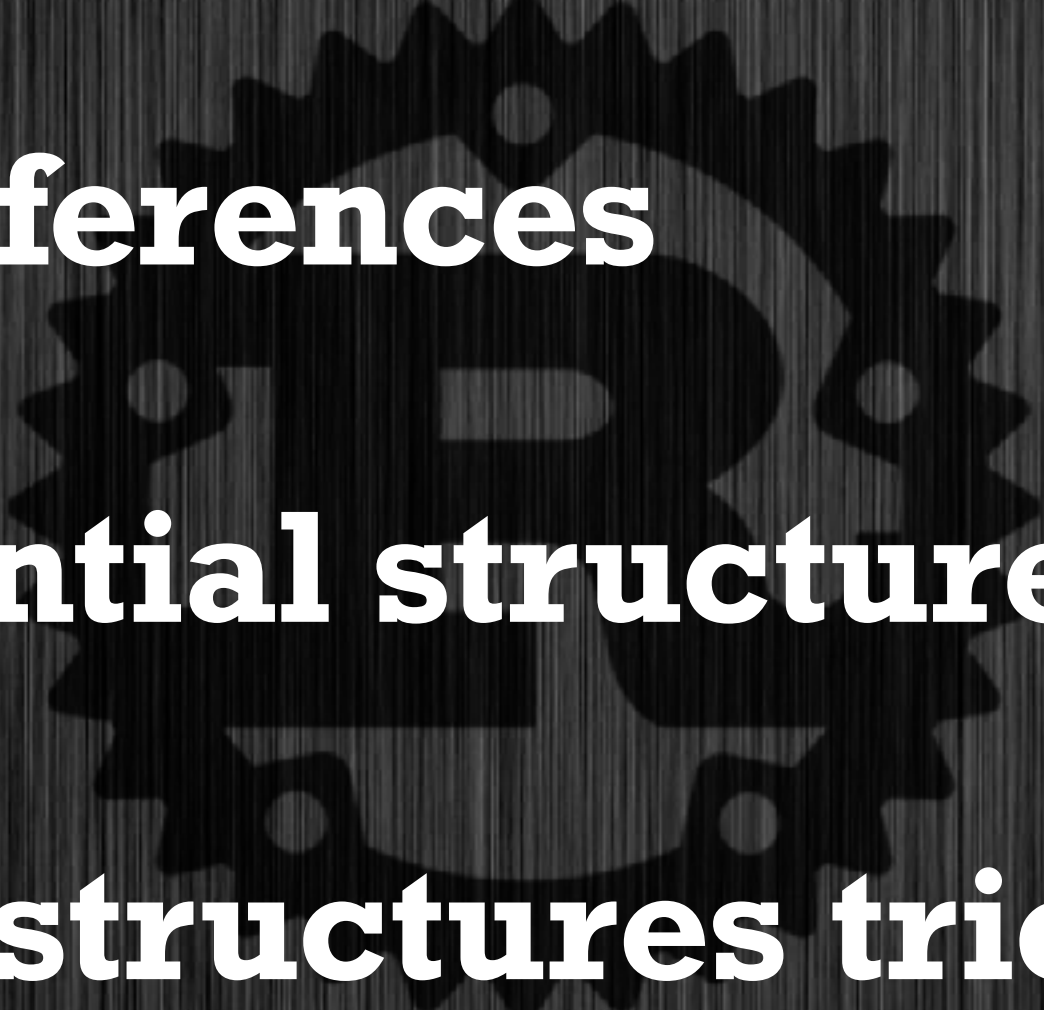


```
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/examples/move`
Hello, world!!
```

memory model: ownership

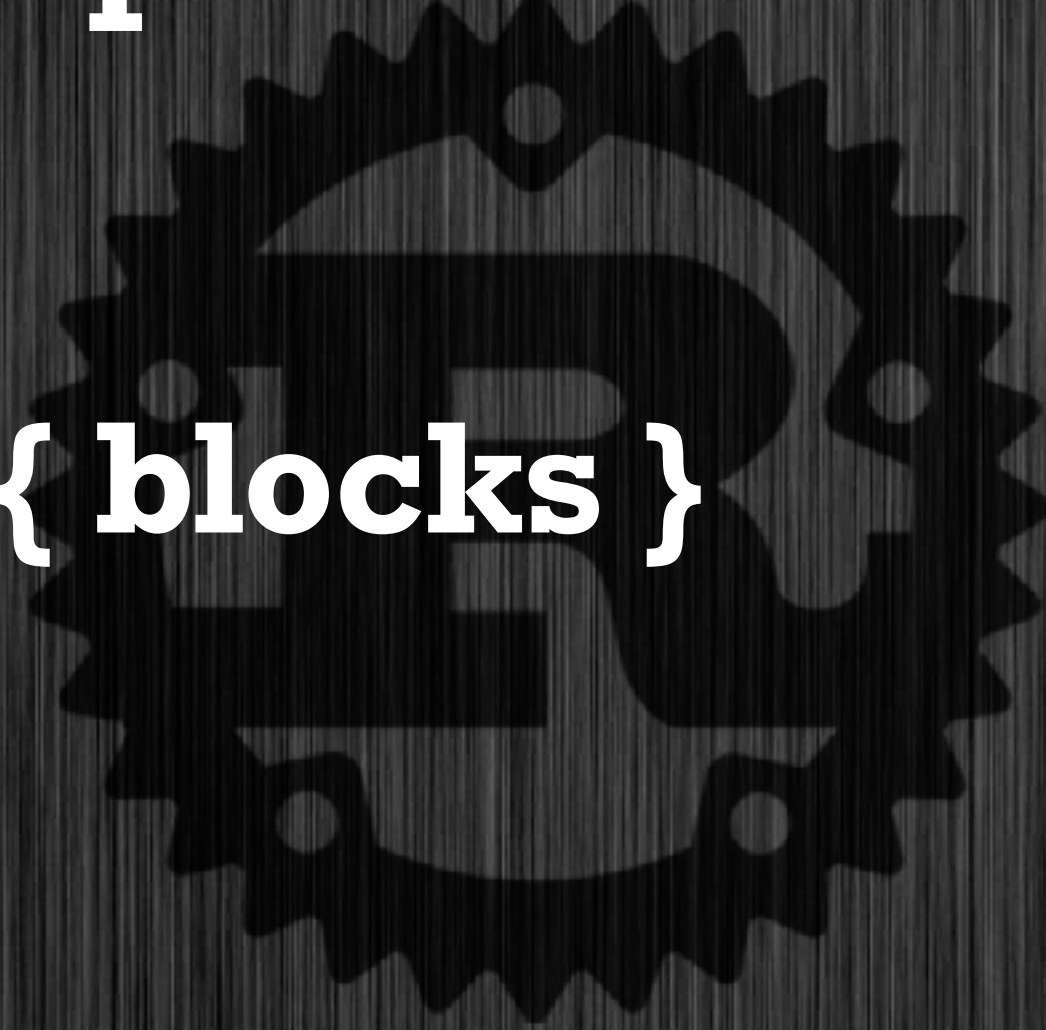
- **multiple immutable references**
OR
 - **a single mutable reference**
 - **shared ownership:**
runtime reference counters in standard library
- 

memory model: ownership

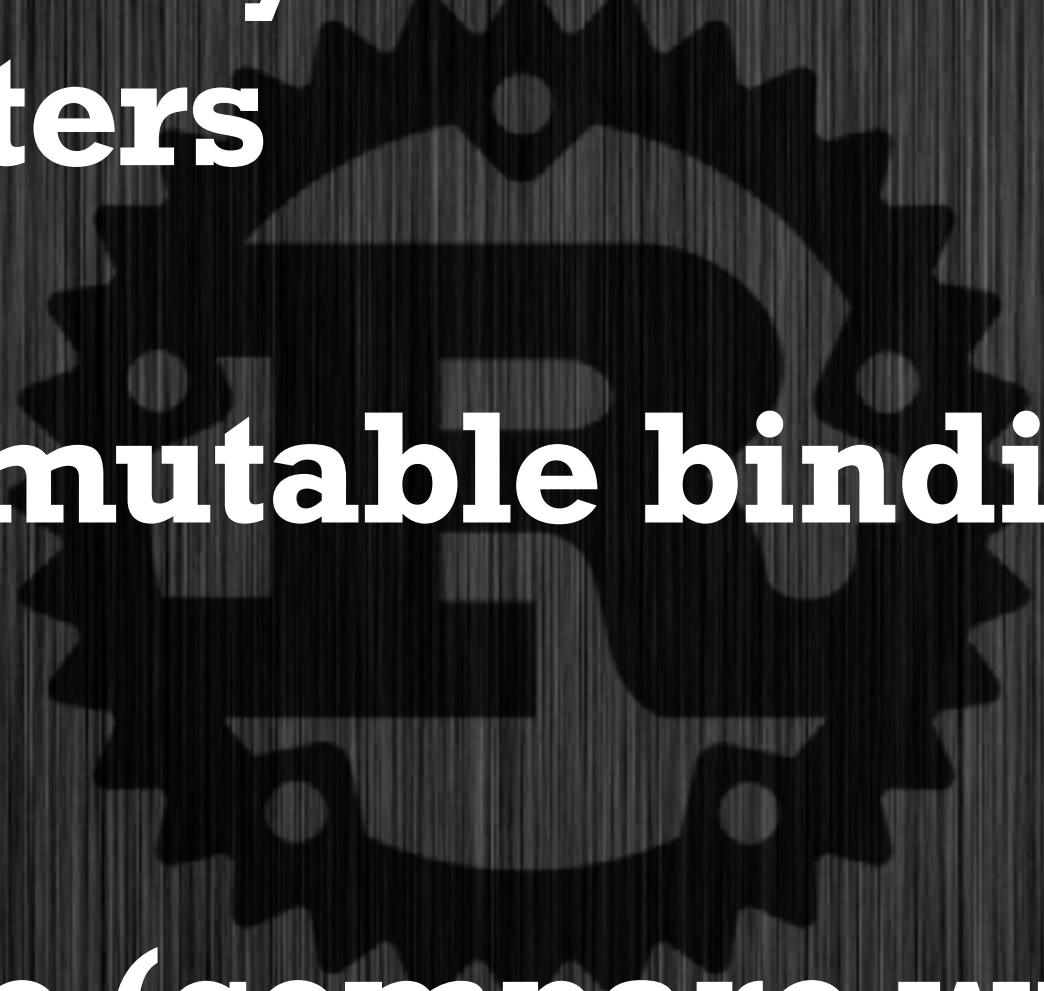
- **no raw pointers (unless you really need them)**
 - **no (simple) cyclical references**
 - **no (simple) self-referential structures**
 - **this makes some data structures tricky (doubly-linked lists, graphs)**
- 

memory model: living with the borrow checker

- **avoid storing references in structs**
 - **pass them as method parameters instead**
- **limit reference scope**
 - **e.g. contain them in { blocks }**
- **give up**
 - **.clone()**
 - **Arc<Mutex<T>>**



memory model: mutability

- **all bindings immutable by default
even function parameters**
 - **`let mut` introduces a mutable binding**
 - **rebinding is allowed
even in the same scope (compare with -Wshadow)**
- 

mutable bindings vs rebinding

```
fn mut_bindings() {  
    let mut i = 1;  
    println!("initially, i = {}", i);  
    {  
        i = i + 1;  
        println!("inside block, i = {}", i);  
    }  
    println!("outside block, i = {}", i);  
}
```

```
fn rebinding() {  
    let i = 1;  
    println!("initially, i = {}", i);  
    {  
        let i = i + 1;  
        println!("inside block, i = {}", i);  
    }  
    println!("outside block, i = {}", i);  
    let i = i + 1;  
    println!("after rebinding, i = {}", i);  
}
```

memory model: ownership

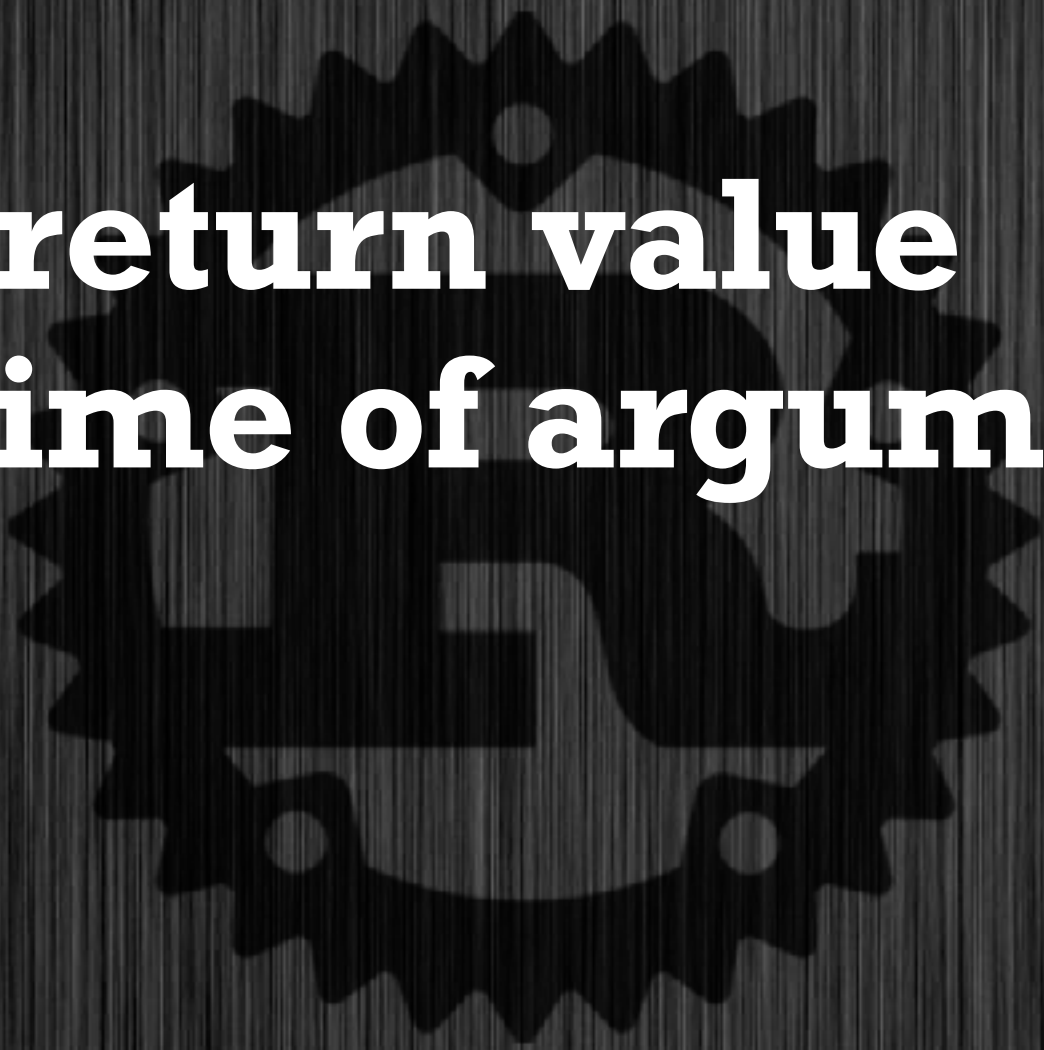
<https://rust-unofficial.github.io/too-many-lists/>



pass/return references

```
fn substr_ref(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}
```

**lifetime of return value
tied to lifetime of argument**



explicit vs elided lifetimes

```
fn substr_ref(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}
```

```
fn substr_ref<'a>(s: &'a str, pos: usize, len: usize) -> &'a str
{
    &s[pos .. pos+len]
}
```

explicit vs elided lifetimes

```
fn substr_ref(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}
```

```
fn substr_ref<'a>(s: &'a str, pos: usize, len: usize) -> &'a str
{
    &s[pos .. pos+len]
}
```

```
fn find<'a, 'b>(haystack: &'a str, needle: &'b str) -> Option<&'a str>
{
    // ...
}
```

```
fn choose<'a>(left: &'a str, right: &'a str) -> &'a str
{
    // ...
}
```

reference vs clone

```
fn substr(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}

fn consume(_: String)
{
}

fn main()
{
    let hello_world = String::from("Hello, world");
    let hello = substr(&hello_world, 0, 5);

    let hello_clone = hello.to_string();

    println!("{}", hello);
    consume(hello_world);
    println!("{}", hello_clone);
}
```

reference vs clone

```
fn substr(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}

fn consume(_: String)
{
}

fn main()
{
    let hello_world = String::from("Hello, world");
    let hello = substr(&hello_world, 0, 5);

    let hello_clone = hello.to_string();

    println!("{}", hello);
    consume(hello_world);
    println!("{}", hello_clone);
}
```

lifetime of `hello\_world`

reference vs clone

```
fn substr(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}

fn consume(_: String)
{
}

fn main()
{
    let hello_world = String::from("Hello, world");
    let hello = substr(&hello_world, 0, 5);

    let hello_clone = hello.to_string();

    println!("{}", hello);
    consume(hello_world);
    println!("{}", hello_clone);
}
```

lifetime of `hello`

reference vs clone

```
fn substr(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}

fn consume(_: String)
{
}

fn main()
{
    let hello_world = String::from("Hello, world");
    let hello = substr(&hello_world, 0, 5);

    let hello_clone = hello.to_string();

    println!("{}", hello);
    consume(hello_world);
    println!("{}", hello_clone);
}
```

**an owned value has its own
lifetime and memory
allocation**

lifetime of `hello\_clone`

reference vs clone

Java 6: original and substring share memory

```
fn substr_ref(s: &str, pos: usize, len: usize) -> &str
{
    &s[pos .. pos+len]
}
```

Java 7: substring is copied into a new allocation

```
fn substr_owned(s: &str, pos: usize, len: usize) -> String
{
    s[pos .. pos+len].to_string()
}
```

`Copy` vs move

```
fn print(i: i32)
{
    println!("{}", i);
}
```

```
fn main()
{
    let i = 10;
    print(i);
    print(i);
}
```

`Copy` vs move

```
fn print(i: i32)
{
    println!("{}", i);
}

fn main()
{
    let i = 10;
    print(i);
    print(i);
}
```

bool

i8/u8

i16/u16

i32/u32

i64/u64

i128/u128

isize/usize

f32

f64

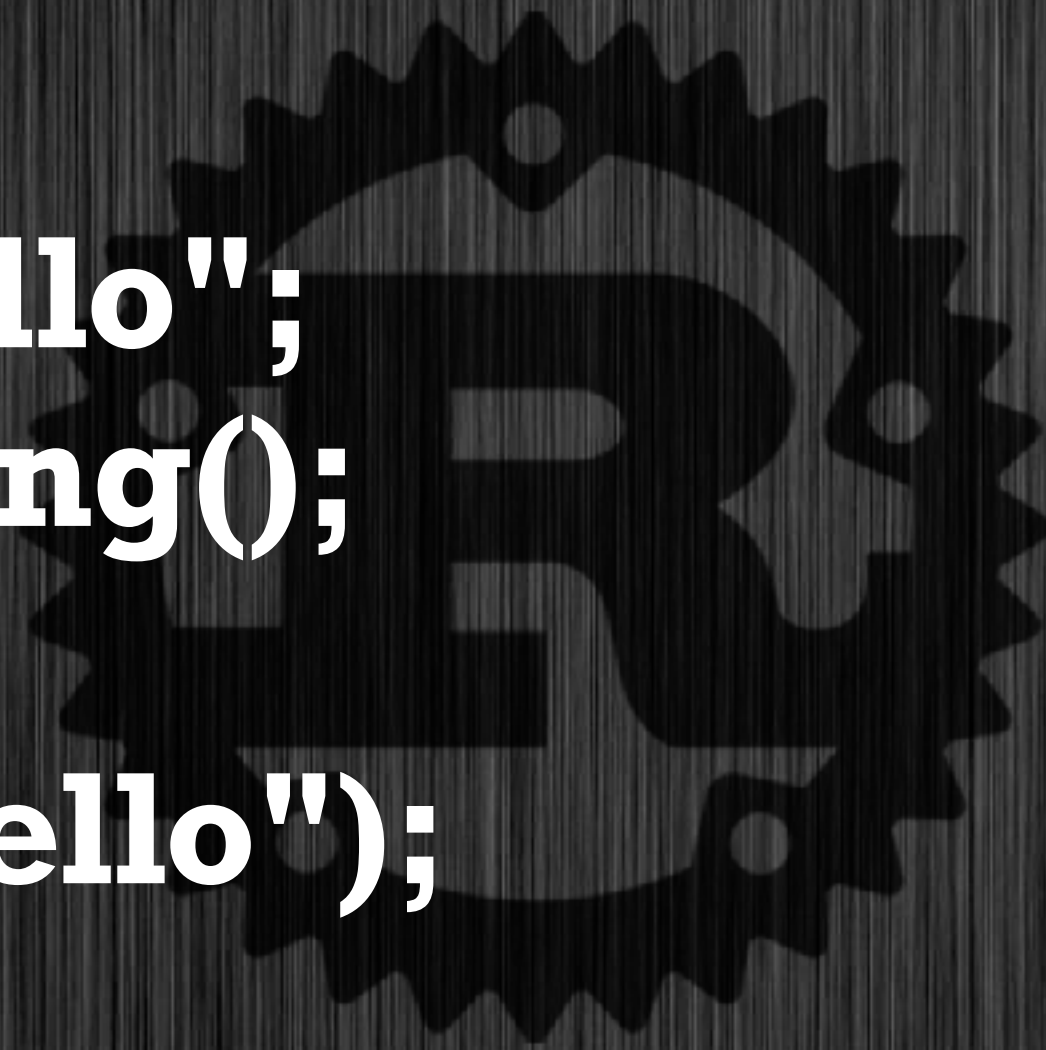
&T

Rust has too many string types!

- **String: owned, mutable, allocated buffer**
- **&String: reference to String, not really used**
- **&str: immutable sequence of characters**
"hello" is a &'static str
C++17 std::string_view, but safer
- **String and &str are always valid UTF-8**

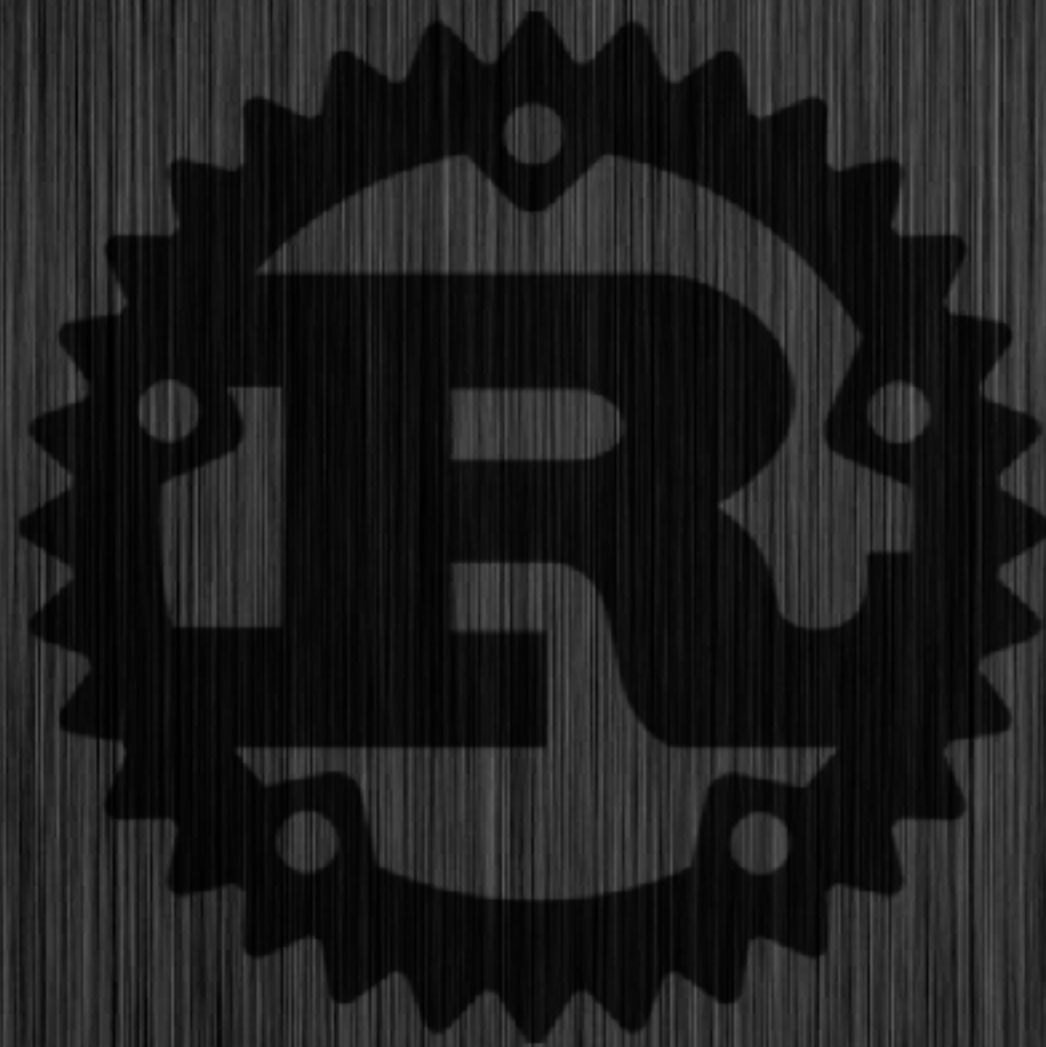
Rust has too many string types!

- **let s: &'static str = "hello";
let owned_s = s.to_string();**
- **let s = String::from("hello");
let ref_s = s.as_str();**



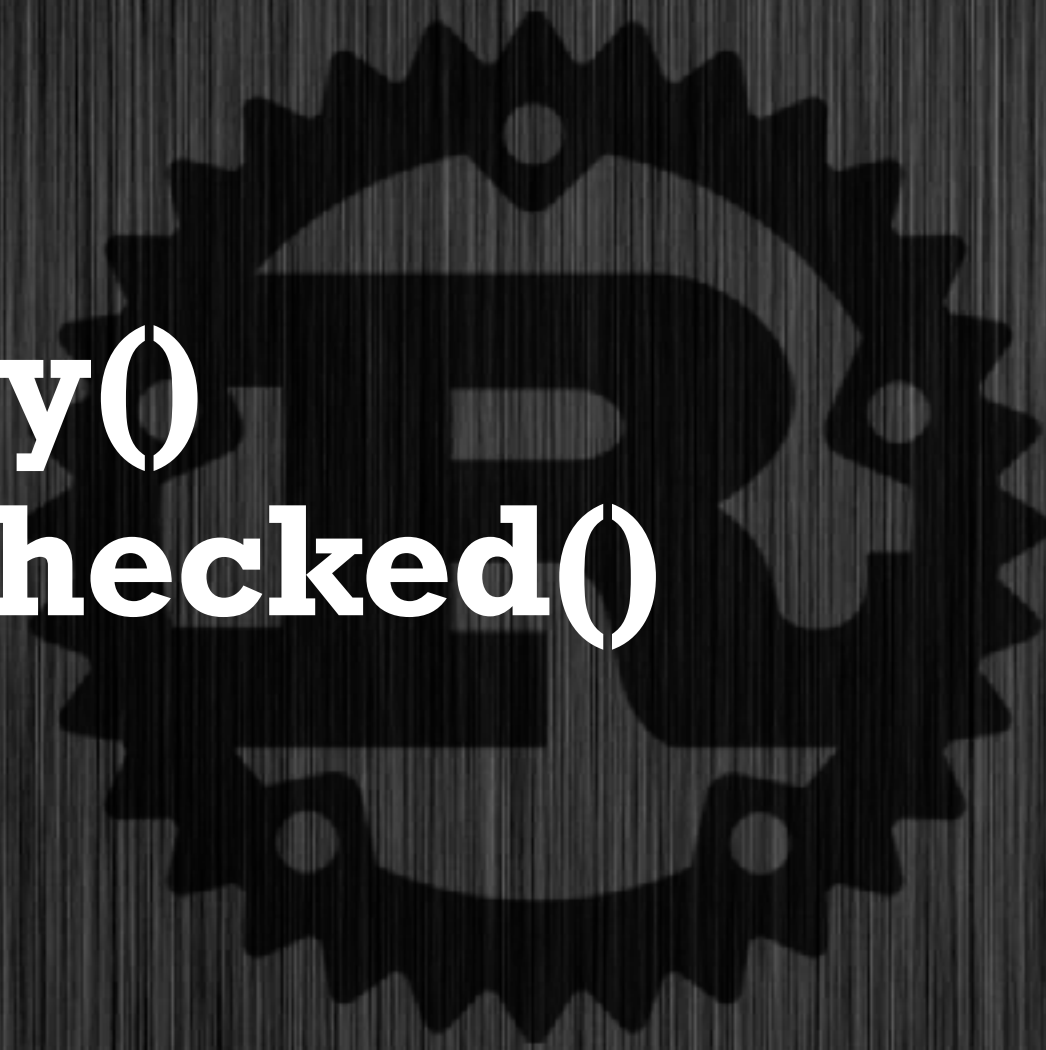
Rust has too many string types!

- **raw byte buffers**
- **`Vec<u8>`**
- **`&Vec<u8>`**
- **`&[u8]`**



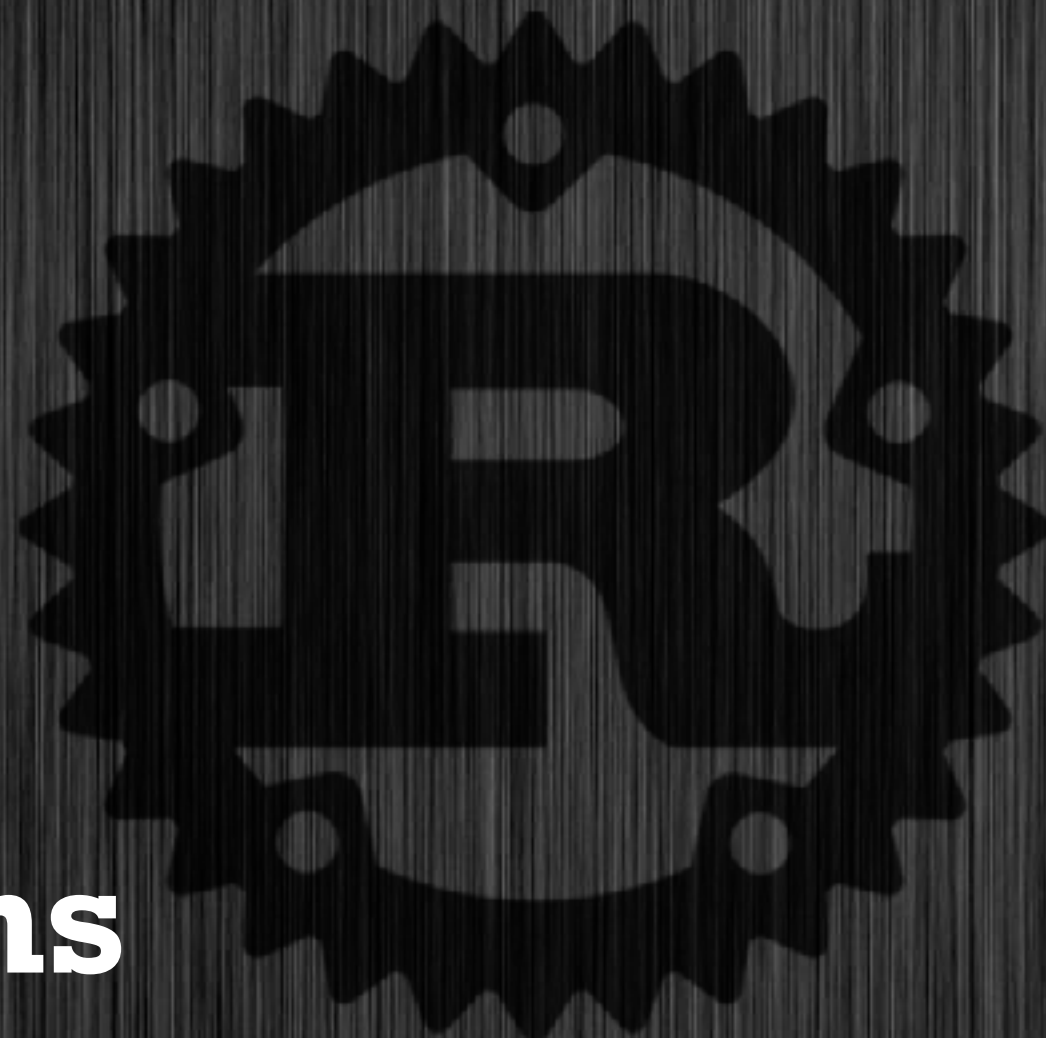
Rust has too many string types!

- **str::from_utf8()**
String::from_utf8()
String::from_utf8_lossy()
String::from_utf8_unchecked()
- **str::as_bytes()**
String::as_bytes()
String::into_bytes()



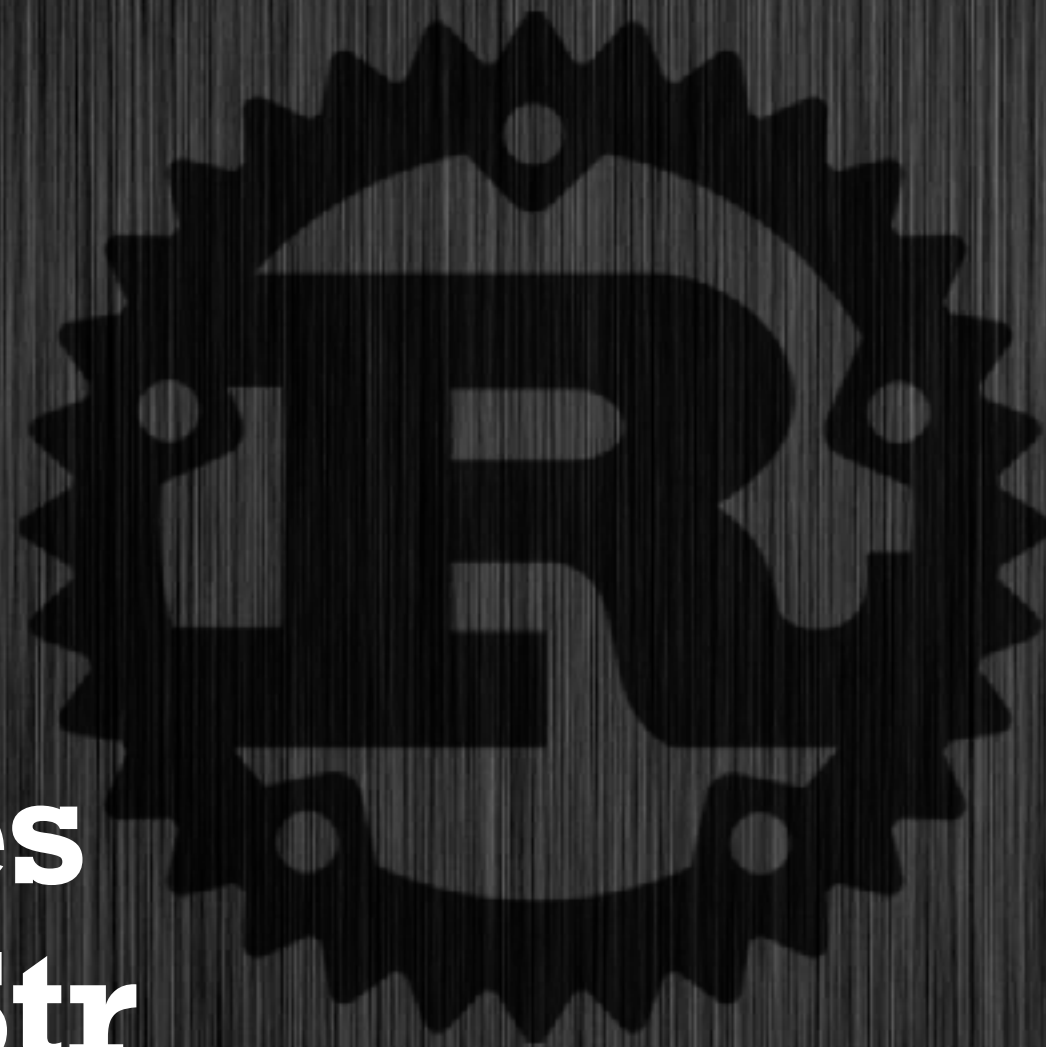
Rust has too many string types!

- **`std::ffi::OsString`**
`std::ffi::OsStr`
- **not UTF-8 safe**
OS-specific conversions
`std::ffi::{unix, windows}::{OsStringExt, OsStrExt}`



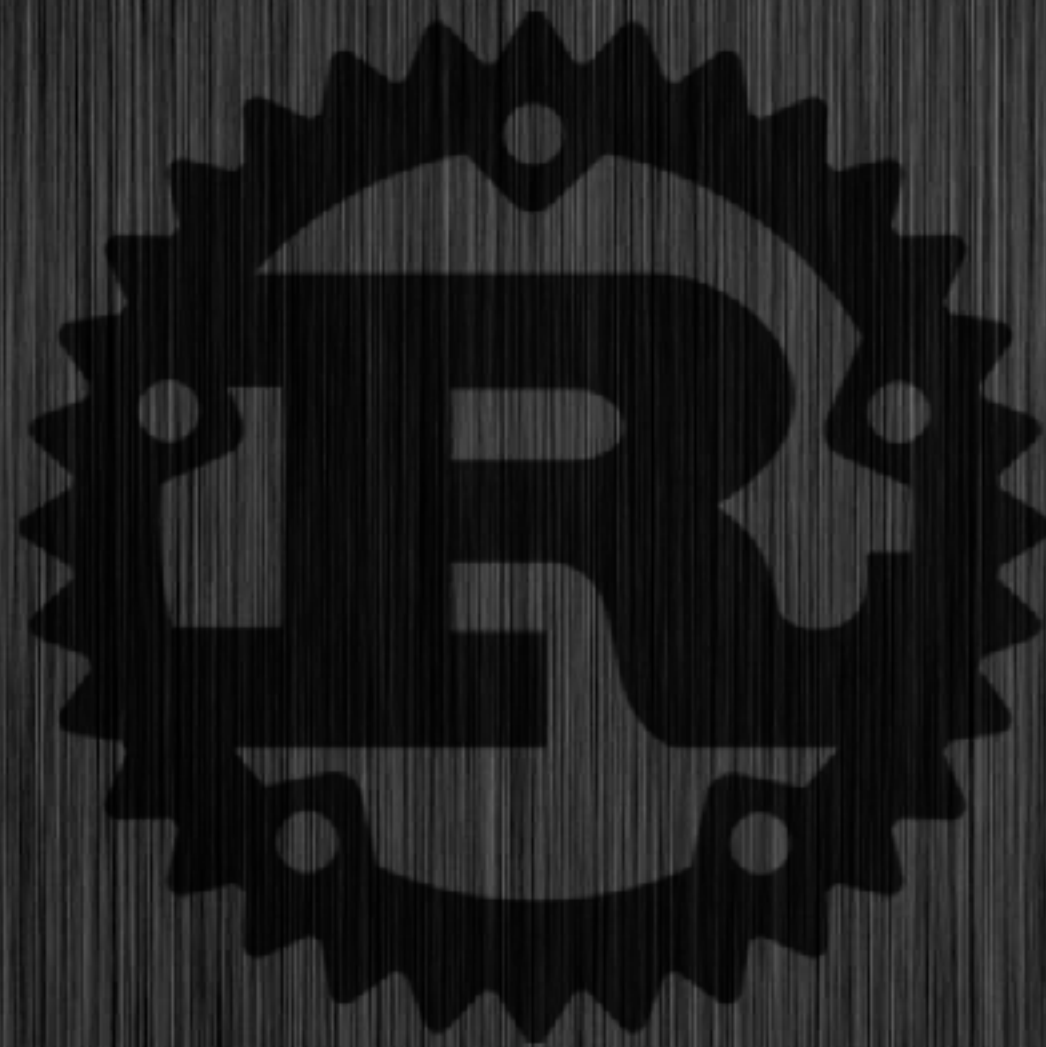
Rust has too many string types!

- **`std::path::PathBuf`**
`std::path::Path`
- **not strictly string types**
wrap an `OsString`/`OsStr`



data structures in Rust

- **struct**
- **tuple**
- **tuple struct**
- **enum**

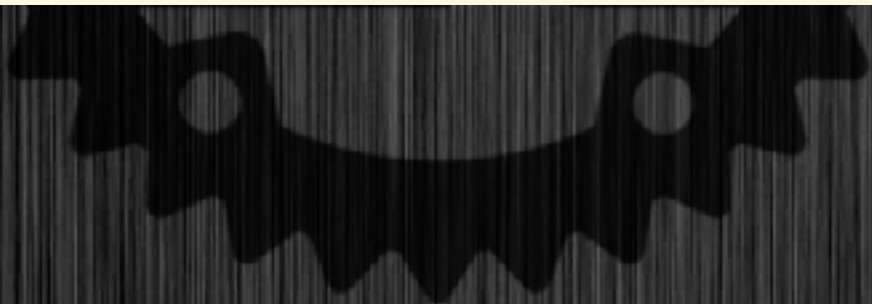


struct



```
struct Rectangle {  
    float width;  
    float height;  
};
```

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}
```



tuple

```
fn tuples() {  
    let corner = (10, 20);  
    let corner: (i32, i32) = (10, 20); // explicit  
  
    println!("coordinates: [{}]", corner.0, corner.1);  
}
```

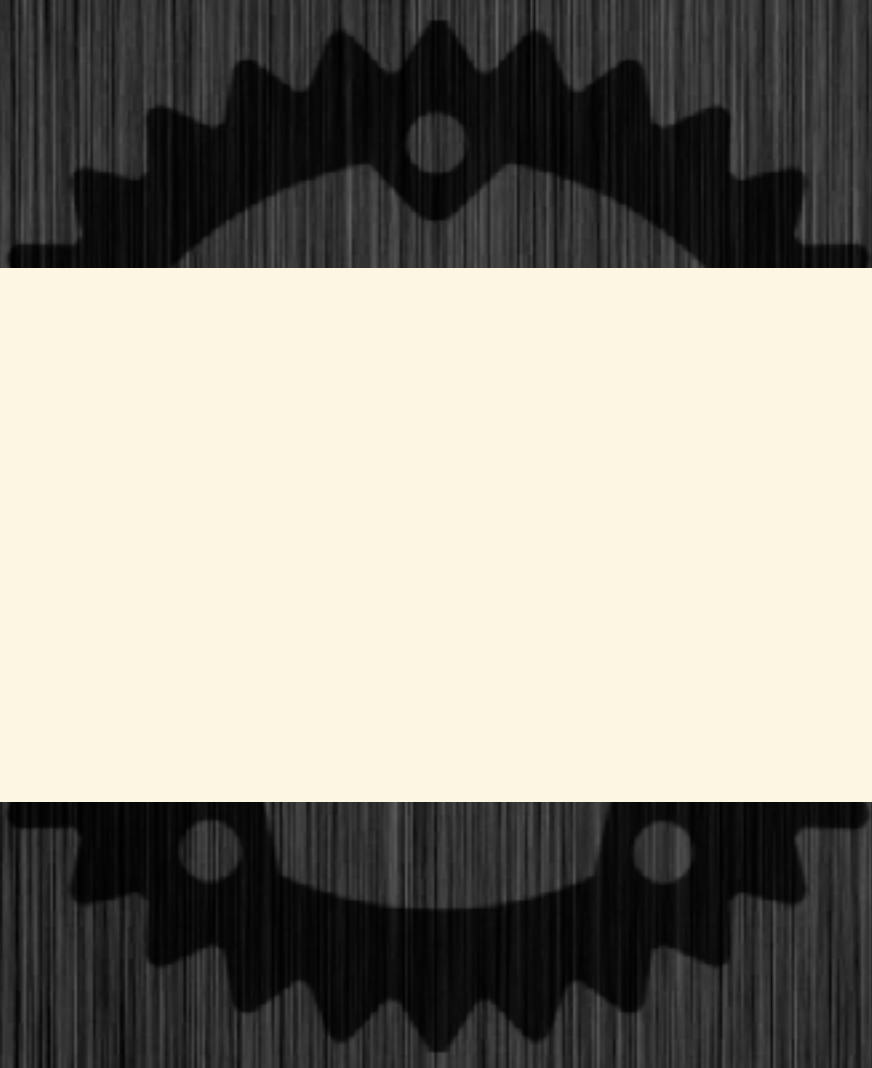
tuple struct

```
struct Point(i32, i32);

fn tuples() {
    let corner = Point(10, 20);

    println!("coordinates: [{}]", corner.0, corner.1);
}
```

enum

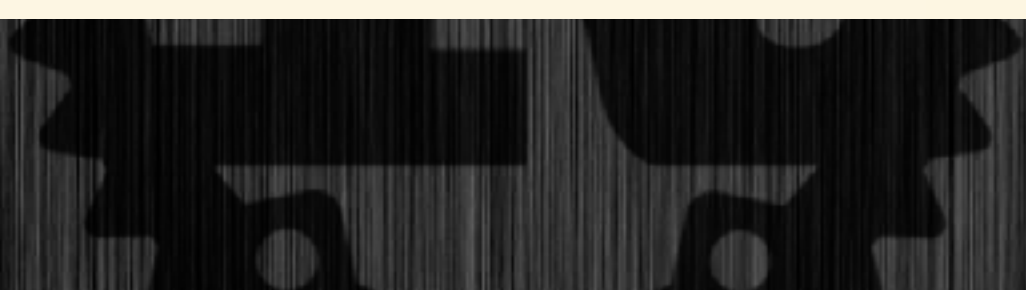


```
enum Color {  
    RED,  
    GREEN,  
    BLUE  
};
```

```
enum Color {  
    Red,  
    Green,  
    Blue,  
}
```

enum

```
enum Color {  
    RGB(f64, f64, f64),  
    HSV(f64, f64, f64),  
    Pantone(String),  
}  
  
let lightblue = Color::RGB(0.5, 0.5, 0.9);
```



```
enum Color {  
    RGB { red: f64, green: f64, blue: f64 },  
    HSV { hue: f64, saturation: f64, value: f64 },  
    Pantone { name: String },  
}  
  
let lightblue = Color::RGB { red: 0.5, green: 0.5, blue: 0.9 };
```

enum + match

```
enum Color {  
    RGB(f64, f64, f64),  
    HSV(f64, f64, f64),  
    Pantone(String),  
}  
  
fn css_name(color: &Color) -> String {  
    match color {  
        Color::RGB(r, g, b) => format!("rgb({}, {}, {})", r, g, b),  
        Color::HSV(h, s, v) => format!("hsl({}, {}, {})", h, s, v),  
        Color::Pantone(name) => pantone_to_css(name),  
    }  
}
```

impl blocks

```
struct Rectangle {  
    float width;  
    float height;  
  
    float get_area() const {  
        return width * height;  
    }  
};
```

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
impl Rectangle {  
    pub fn get_area(&self) -> f64 {  
        self.width * self.height  
    }  
}
```

&self, &mut self, self

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
impl Rectangle {  
    pub fn get_area(&self) -> f64 {  
        self.width * self.height  
    }  
  
    pub fn double_width(&mut self) {  
        self.width *= 2;  
    }  
  
    pub fn into_dimensions(self) -> (f64, f64) {  
        (self.width, self.height)  
    }  
}
```

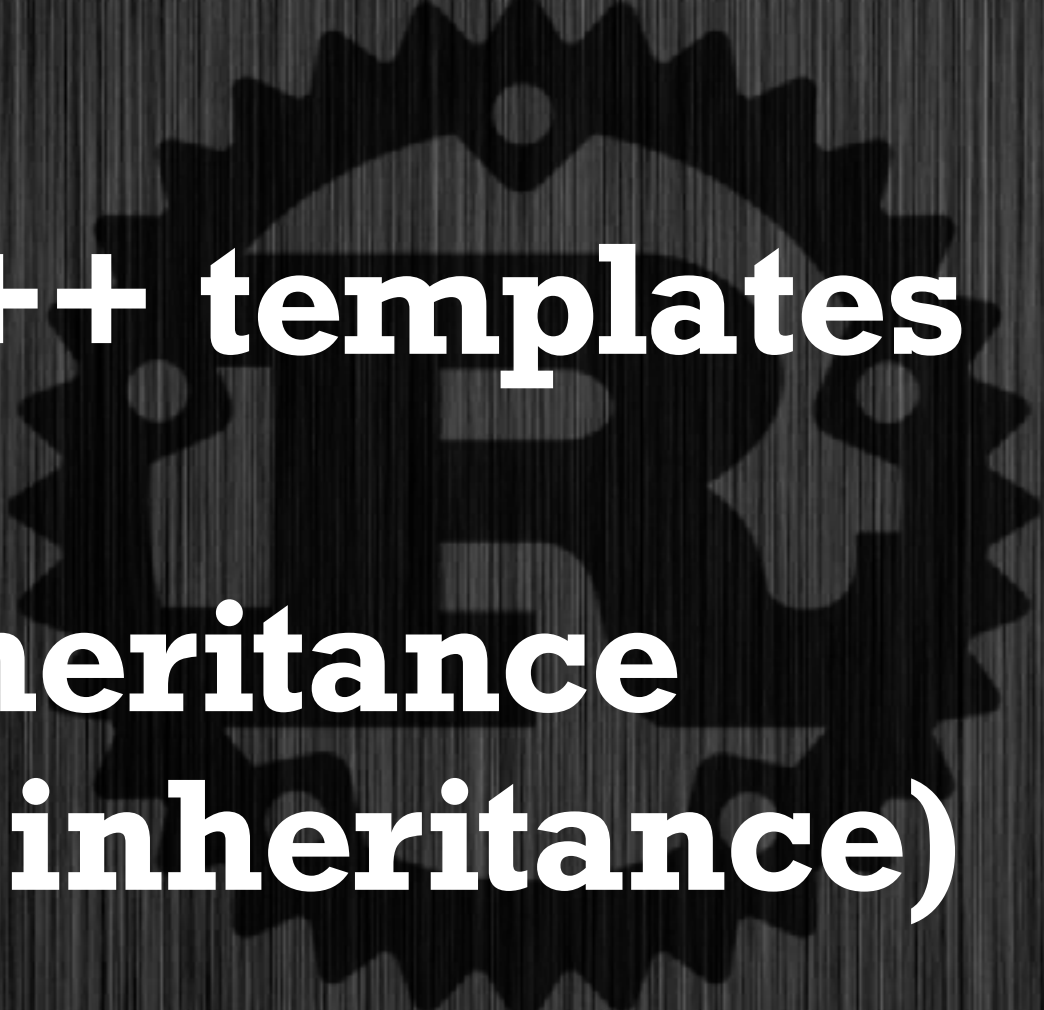
impl blocks work for enums and tuple structs too

```
enum Color {
    RGB(f64, f64, f64),
    HSV(f64, f64, f64),
    Pantone(String),
}

impl Color {
    fn css_name(&self) -> String {
        match self {
            Color::RGB(r, g, b) => format!("rgb({}, {}, {})", r, g, b),
            Color::HSV(h, s, v) => format!("hsl({}, {}, {})", h, s, v),
            Color::Pantone(name) => pantone_to_css(name),
        }
    }
}

let lightblue = Color::RGB(0.5, 0.5, 0.9);
println!("{}", lightblue.css_name());
// rgb(0.5, 0.5, 0.9)
```

traits and generic programming

- **traits: similar to Java interfaces or C++20 concepts**
 - **generics: similar to C++ templates**
 - **no implementation inheritance**
 - **trait impls (interface inheritance)**
 - **composition**
- 

traits and generic programming



Rust is not OOP
(but you can often pretend)

think STL, not Qt

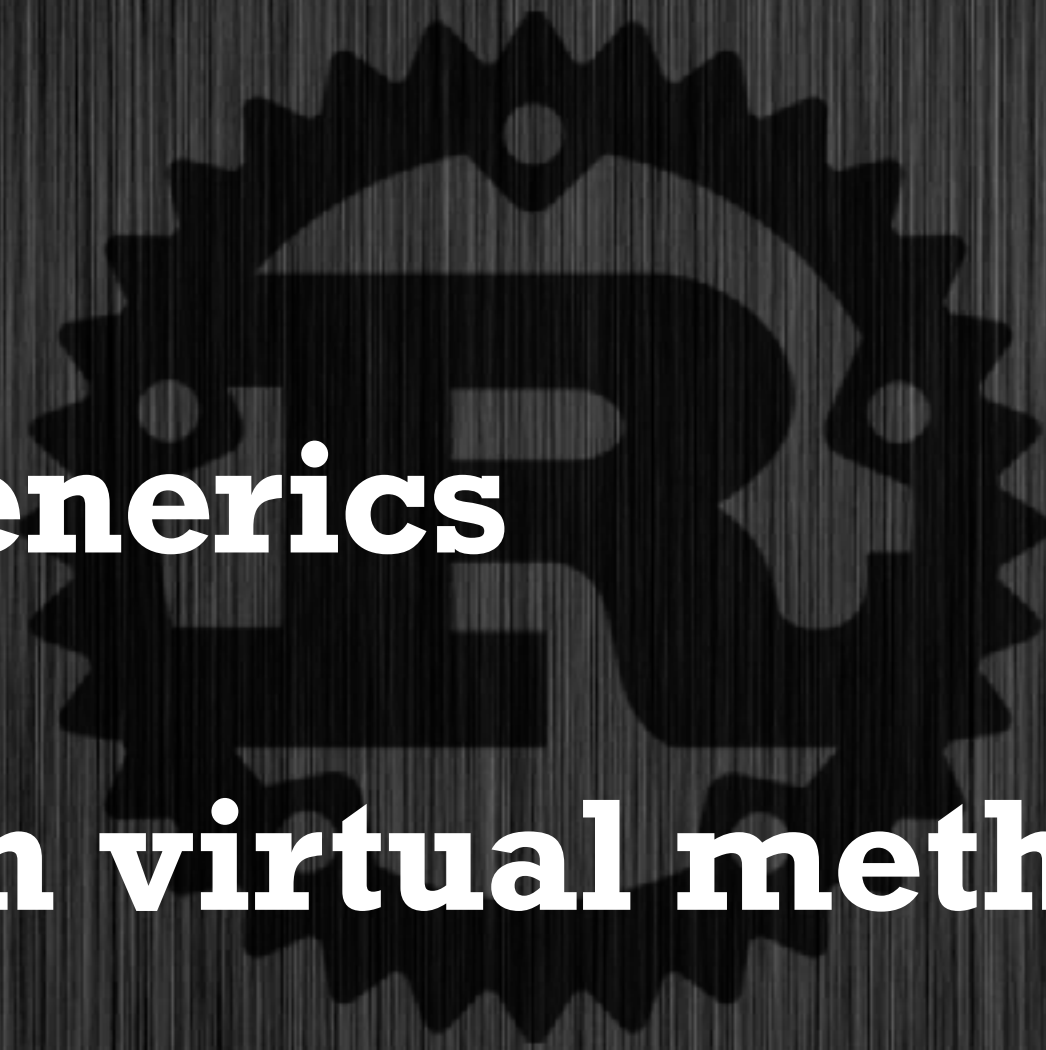
traits and generic programming

```
struct Rectangle {  
    float width;  
    float height;  
  
    float get_area() const {  
        return width * height;  
    }  
};  
  
struct Circle  
{  
    float radius;  
  
    float get_area() const {  
        return 3.14 * radius * radius;  
    }  
};
```

traits and generic programming

C++

- **static dispatch with generics**
- **dynamic dispatch with virtual methods**



C++ dynamic dispatch with inheritance

```
struct Shape {  
    virtual float get_area() const = 0;  
}
```

```
struct Rectangle : public Shape {  
    float width;  
    float height;  
  
    float get_area() const override {  
        return width * height;  
    }  
};
```

```
struct Circle : public Shape  
{  
    float radius;  
  
    float get_area() const override {  
        return 3.14 * radius * radius;  
    }  
};
```

```
void print_area(const Shape* shape)  
{  
    std::cout << shape->get_area() << '\n';  
}
```

C++ static dispatch with templates

```
struct Rectangle {  
    float width;  
    float height;  
  
    float get_area() const {  
        return width * height;  
    }  
};  
  
struct Circle  
{  
    float radius;  
  
    float get_area() const {  
        return 3.14 * radius * radius;  
    }  
};
```

```
template<class S>  
void print_area(const S* shape)  
{  
    std::cout << shape->get_area() << '\n';  
}
```

C++ static dispatch with templates

```
class LandWarInAsia {  
    // ...  
  
    // send troops to capture the chosen area  
    bool get_area();  
}
```

```
template<class S>  
void print_area(const S* shape)  
{  
    std::cout << shape->get_area() << '\n';  
}
```

oops

traits and generic programming

Rust

- **static dispatch with generics**
- **dynamic dispatch with trait objects**



traits: static dispatch with generics

```
trait Shape {  
    fn get_area(&self) -> f64;  
}
```

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}
```

```
impl Shape for Rectangle {  
    fn get_area(&self) -> f64 {  
        self.width * self.height  
    }  
}
```

```
struct Circle {  
    radius: f64,  
}
```

```
impl Shape for Circle {  
    fn get_area(&self) -> f64 {  
        3.14 * self.radius * self.radius  
    }  
}
```

```
fn print_area<S: Shape>(shape: &S) {  
    println!("{}", shape.get_area());  
}
```

traits: dynamic dispatch with trait objects

```
trait Shape {  
    fn get_area(&self) -> f64;  
}
```

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}
```

```
impl Shape for Rectangle {  
    fn get_area(&self) -> f64 {  
        self.width * self.height  
    }  
}
```

```
struct Circle {  
    radius: f64,  
}
```

```
impl Shape for Circle {  
    fn get_area(&self) -> f64 {  
        3.14 * self.radius * self.radius  
    }  
}
```

```
fn print_area(shape: &dyn Shape) {  
    println!("{}", shape.get_area());  
}
```

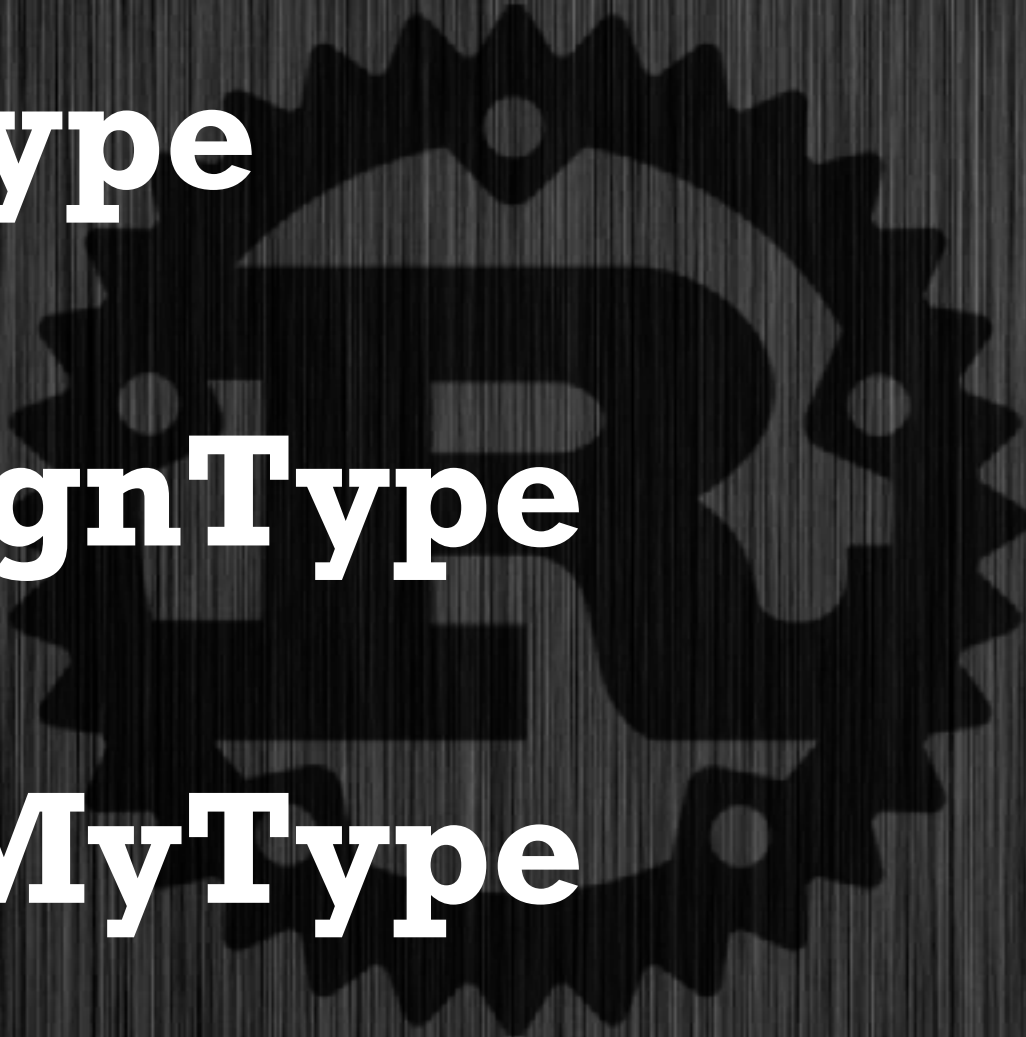
traits: disambiguation

```
impl Shape for Continent {  
    fn get_area(&self) -> f64 { ... }  
}
```

```
impl LandWar for Continent {  
    fn get_area(&mut self) { ... }  
}
```

```
let mut asia = Continent::new(...);  
// geometry  
Shape::get_area(&asia)  
// warfare  
LandWar::get_area(&mut asia)  
  
// no ambiguity here  
fn print_area<S: Shape>(shape: &S) {  
    println!("{}", shape.get_area());  
}
```

traits and generic programming

- **impl MyTrait for MyType**
 - **impl MyTrait for ForeignType**
 - **impl ForeignTrait for MyType**
- 

traits: default impls

```
trait Shape {  
    fn get_area(&self) -> f64;  
  
    fn print_area(&self) {  
        println!("{}", self.get_area());  
    }  
}  
  
// ...  
some_rectangle.print_area();  
// ...
```

traits: blanket impls

```
use std::io::Write;

trait WriteHelloWorld {
    fn write_hello_world(&mut self) -> std::io::Result<usize>;
}

impl<W: Write> WriteHelloWorld for W {
    fn write_hello_world(&mut self) -> std::io::Result<usize> {
        self.write("Hello, world\n".as_bytes())
    }
}
```

traits: blanket impls

```
use std::io::Write;

trait WriteHelloWorld {
    fn write_hello_world(&mut self) -> std::io::Result<usize>;
}

impl<W: Write> WriteHelloWorld for W {
    fn write_hello_world(&mut self) -> std::io::Result<usize> {
        self.write("Hello, world\n".as_bytes())
    }
}

impl WriteHelloWorld for std::io::Stdout {
    fn write_hello_world(&mut self) -> std::io::Result<usize> {
        self.write("Hello, stdout world\n".as_bytes())
    }
}
```

not yet supported in stable Rust

generic data structures

```
enum Color<C> {  
    RGB(C, C, C),  
    HSV(C, C, C),  
    Pantone(String),  
}
```



generic data structures

```
enum Color<C> {  
    RGB(C, C, C),  
    HSV(C, C, C),  
    Pantone(String),  
}  
  
impl<C> Color<C> {  
    fn css_name(&self) -> String {  
        match color {  
            Color::RGB(r, g, b) => format!("rgb({}, {}, {})", r, g, b),  
            Color::HSV(h, s, v) => format!("hsl({}, {}, {})", h, s, v),  
            Color::Pantone(name) => pantone_to_css(name),  
        }  
    }  
}
```

generic data structures

```
enum Color<C> {  
    RGB(C, C, C),  
    HSV(C, C, C),  
    Pantone(String),  
}  
  
impl<C: Display> Color<C> {  
    fn css_name(&self) -> String {  
        match color {  
            Color::RGB(r, g, b) => format!("rgb({}, {}, {})", r, g, b),  
            Color::HSV(h, s, v) => format!("hsl({}, {}, {})", h, s, v),  
            Color::Pantone(name) => pantone_to_css(name),  
        }  
    }  
}
```

generic data structures

```
enum Color<C: Display> {  
    RGB(C, C, C),  
    HSV(C, C, C),  
    Pantone(String),  
}  
  
impl<C: Display> Color<C> {  
    fn css_name(&self) -> String {  
        match color {  
            Color::RGB(r, g, b) => format!("rgb({}, {}, {})", r, g, b),  
            Color::HSV(h, s, v) => format!("hsl({}, {}, {})", h, s, v),  
            Color::Pantone(name) => pantone_to_css(name),  
        }  
    }  
}
```

what about constructors?

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
let rect = Rectangle {  
    width: 10,  
    height: 20,  
};
```



what about constructors?

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
impl Rectangle {  
    pub fn new(width: f64, height: f64) -> Rectangle {  
        Rectangle {  
            width: width,  
            height: height,  
        }  
    }  
}
```

what about constructors?

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
impl Rectangle {  
    pub fn new(width: f64, height: f64) -> Self {  
        Self {  
            width: width,  
            height: height,  
        }  
    }  
}
```

what about constructors?

```
struct Rectangle {  
    pub width: f64,  
    pub height: f64,  
}  
  
impl Rectangle {  
    pub fn new(width: f64, height: f64) -> Self {  
        Self {  
            width,  
            height,  
        }  
    }  
}
```

what about constructors?

```
struct Rectangle {  
    width: f64,  
    height: f64,  
}  
  
impl Rectangle {  
    pub fn new(width: f64, height: f64) -> Self {  
        Self {  
            width,  
            height,  
        }  
    }  
}
```

`new` is just a convention

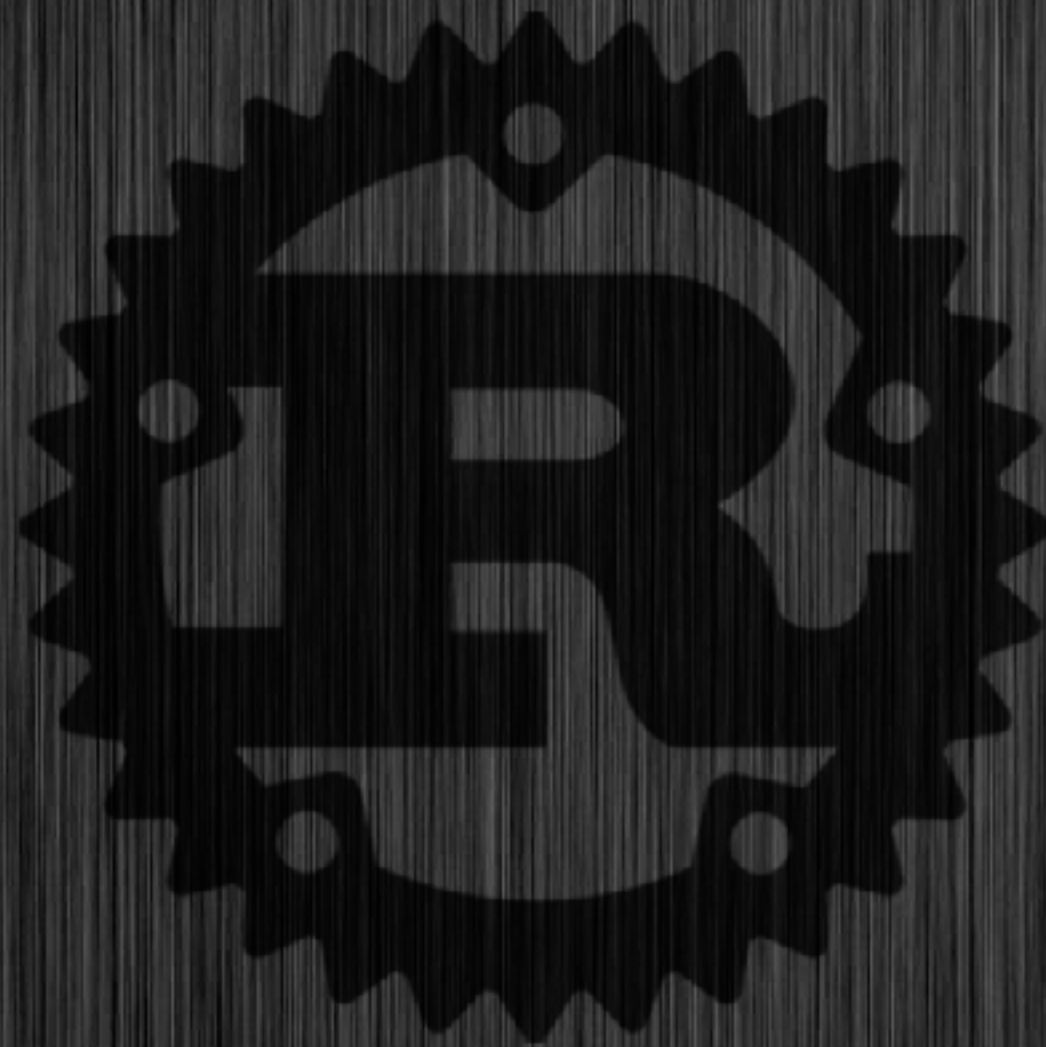
```
impl File {  
    /// open an existing file  
    pub fn open(path: &str) -> Self {  
        // ...  
    }  
  
    /// create a new file  
    pub fn create(path: &str) -> Self {  
        // ...  
    }  
}
```

destructors?

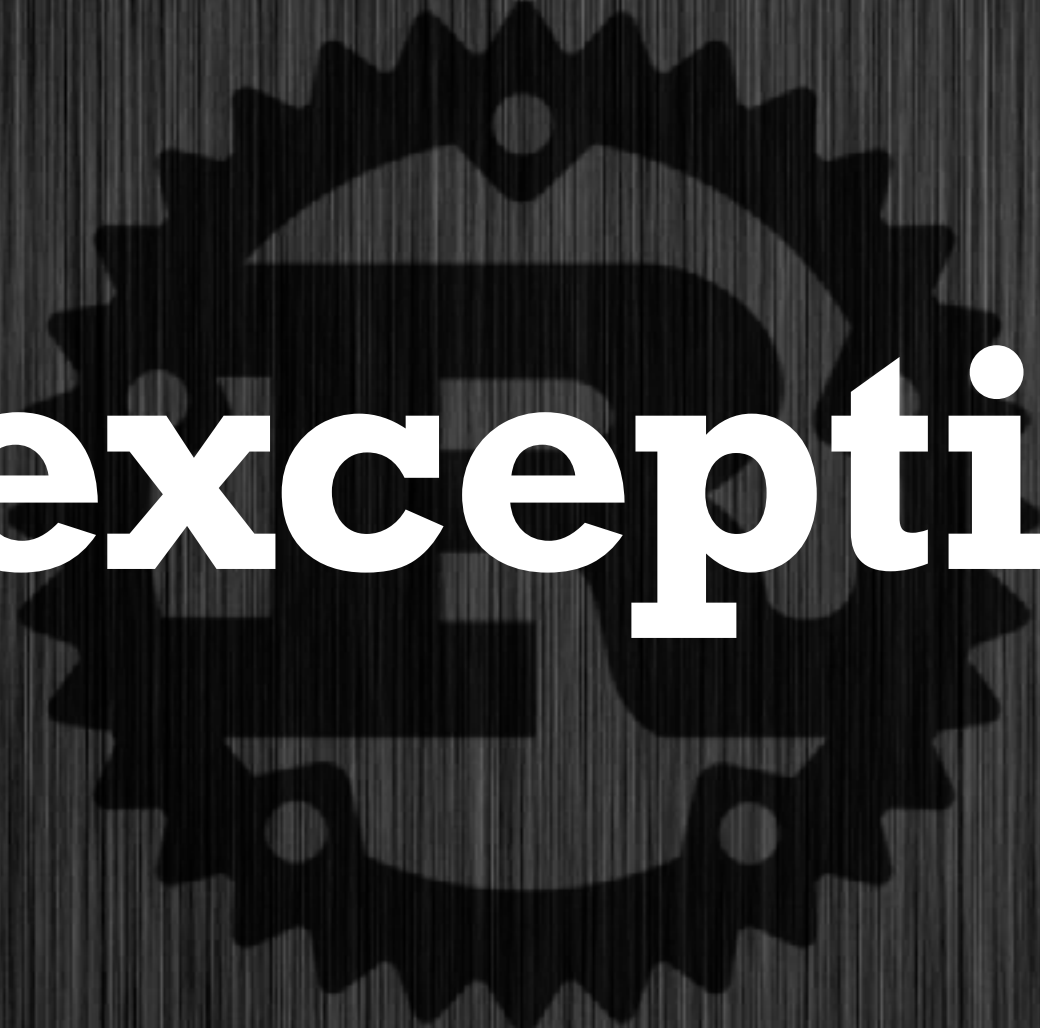
```
impl Drop for File {  
    fn drop(&mut self) {  
        self.fd.close();  
    }  
}
```

Option, Result and error handling

- **Option<T>**
- **Result<T, E>**
- **the ``?`** operator
- **panic!**
std::panic::catch_unwind if you **really** need it



Option, Result and error handling



no exceptions

Option<T>, fixing the billion dollar mistake

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

... and a million useful methods

Option<T>

```
class logger {  
    // ...  
    void set_prefix(const std::string& prefix);  
    // ...  
    std::string m_prefix;  
}  
  
void logger::set_prefix(const std::string& prefix)  
{  
    if (!prefix.empty()) {  
        m_prefix = prefix + ": ";  
    } else {  
        m_prefix = "";  
    }  
}
```

Option<T>

```
struct Logger {  
    // ...  
    prefix: String,  
    // ...  
}  
  
impl Logger {  
    pub fn set_prefix(&mut self, prefix: Option<&str>) {  
        if prefix.is_some() {  
            let prefix = prefix.unwrap();  
            self.prefix = format!("{}: ", prefix);  
        } else {  
            self.prefix = String::new();  
        }  
    }  
}
```

Option<T>

```
struct Logger {  
    // ...  
    prefix: String,  
    // ...  
}  
  
impl Logger {  
    pub fn set_prefix(&mut self, prefix: Option<&str>) {  
        self.prefix = match prefix {  
            Some(prefix_str) => format!("{}", prefix_str),  
            None => String::new(),  
        }  
    }  
}
```

if let

```
struct Logger {  
    // ...  
    pub prefix: Option<String>,  
    // ...  
}  
  
impl Logger {  
    pub fn log(&self, msg: &str) {  
        if let Some(ref prefix) = self.prefix {  
            print!("{}", prefix);  
        }  
  
        println!("{}", msg);  
    }  
}
```

Result<T, E>

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

Result<T, E>

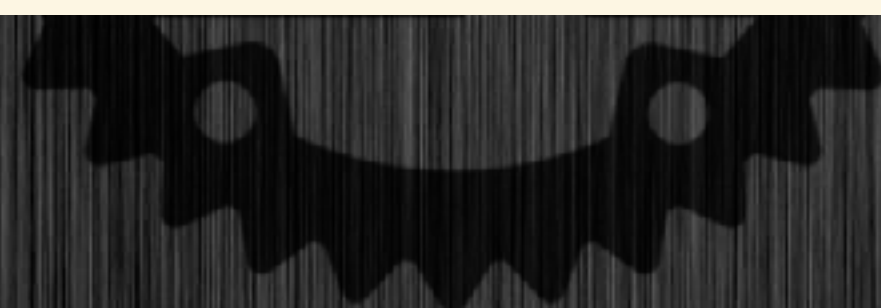
```
impl File {  
    /// open an existing file  
    pub fn open(path: &str) -> Result<Self, std::io::Error> {  
        // ...  
    }  
  
    /// create a new file  
    pub fn create(path: &str) -> Result<Self, std::io::Error> {  
        // ...  
    }  
}
```

Result<T, E>

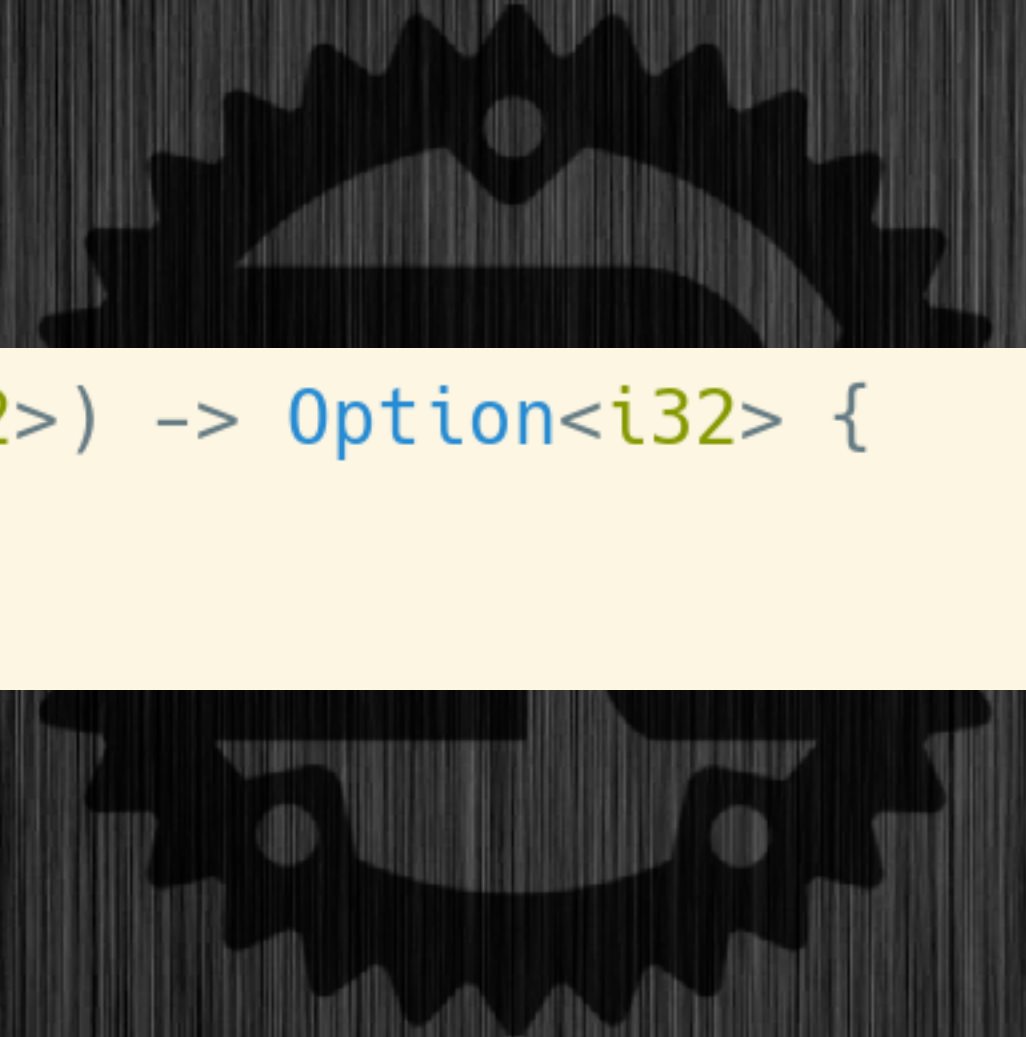
```
fn open_log_file(path: &str) -> Result<Logger, std::io::Error> {  
    let file = match File::create(path) {  
        Ok(file) => file,  
        Err(e) => return Err(e),  
    }  
  
    Ok(Logger::from(file))  
}
```



```
fn open_log_file(path: &str) -> Result<Logger, std::io::Error> {  
    let file = File::create(path)?;  
    Ok(Logger::from(file))  
}
```



? also works with Option<T>



```
fn maybe_add(a: Option<i32>, b: Option<i32>) -> Option<i32> {  
    Some(a? + b?)  
}
```

```
#include <iostream>

void fizzbuzz(unsigned int n)
{
    // ...
}

int main()
{
    unsigned int n;

    std::cout << "Input a number\n";
    std::cin >> n;
    std::cout << "FizzBuzz from 1 to " << n << '\n';

    fizzbuzz(n);
}
```

what happens when:

- **you can't read from stdin?**
- **what you read was not a number?**

```
use std::io::stdin;
use std::error::Error;

fn fizzbuzz(n: usize) {
    // ...
}

fn main() {
    println!("Input a number");
    let mut s = String::new();
    stdin().read_line(&mut s);
    let n = s.parse::<usize>();

    println!("FizzBuzz from 1 to {}", n);
    fizzbuzz(n);
}
```

naive conversion to Rust



```
use std::io::stdin;
use std::error::Error;

fn fizzbuzz(n: usize) {
    // ...
}

fn main() -> Result<(), Box<dyn Error>> {
    println!("Input a number");
    let mut s = String::new();
    stdin().read_line(&mut s)?;
    let n = s.parse::<usize>()?;

    println!("FizzBuzz from 1 to {}", n);
    fizzbuzz(n);
    Ok(())
}
```

fix compile errors

- **read_line() and parse() return Results**
- **just propagate them out of main()**
- **... which will panic with an optional stack trace**

```
use std::io::stdin;
use std::error::Error;

fn fizzbuzz(n: usize) {
    // ...
}

fn main() -> Result<(), Box<dyn Error>> {
    println!("Input a number");
    let mut s = String::new();
    stdin().read_line(&mut s)?;
    let n = s.trim_end().parse::<usize>()?;

    println!("FizzBuzz from 1 to {}", n);
    fizzbuzz(n);
    Ok(())
}
```

... and a runtime failure:

**we need to trim the '\n' from
the string**

**"123\n" doesn't strictly
represent a number**