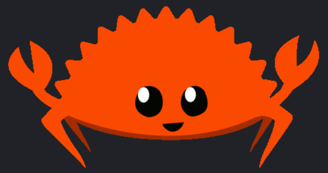
Rust Lang



Tyler Murphy

Rust Book 🚀

https://doc.rust-lang.org/book

What Is Rust?

- Systems Language
- Performant
- Memory Safety
- Fearless Concurrency

Memory Safety and Concurrency is Hard

With Normal Programs

- Segmentation fault (core dumped)
- Undesired behavior
- Instability

Rust Superpowers

- Rich Errors
- Borrow Checker
- Fearless Concurrency
- No NULL
- Macros
- Tests

Rich Errors

No Errors (Javascript)

```
let cool_array = ['let', 'me', 'interject']
console.log(cool_array[5])

// No errors
// Prints undefined
```

Bad Errors (Java)

```
String getValue(String key){
    return map.get(key);
}
```

```
getValue(null)
```

```
java.lang.NullPointerException
```

Rich Errors (Rust)

```
fn main() {
     let cool_array = ["for", "a", "moment"];
     println!("{}", cool_array[5])
}
```

```
$ cargo build
    Compiling thing v0.1.0 (/home/tylerm/Documents/rust/thing)
error: this operation will panic at runtime
 --> src/main.rs:4:20
  |
4 |     println!("{}", cool_array[5])
  |                    ^^^^^^^^^^^^^ index out of bounds: the length is 3 but the index is 5
  |
```

Rust Tells You What To Do

```
$ cargo build
   Compiling thing v0.1.0 (/home/tylerm/Documents/rust/thing)
error[E0308]: mismatched types
 --> src/main.rs:6:10
  |
6 |     func("foobar");
  |     ---- ^^^^^^^^- help: try using a conversion method: `.to_string()`
  |     |    |
  |     |    expected struct `String`, found `&str`
  |     arguments to this function are incorrect
  |
```

Borrow Checker

(🚀)

Memory Management

- No Garbage Collection
- No Pointers

Variable Ownership

- Variables are immutable (by default)
- Variables can only be owned by one thing
- Variables are dropped as soon as their ownership ends

Source

```rust
fn main() {
    let opinion = "Vim is better than Emacs";
    println!("{}", opinion);
}
```

Compiled

```rust
fn main() {
    let opinion = "Vim is better than Emacs";
    println!("{}", opinion);

    drop(opinion); // added here by the rust compiler
}
```

Source

```
fn main() {

    let opinion = Opinion("Systemd is bad".to_string());
    thing(opinion)

}

fn thing(s: Opinion) {

    println!("{}", s.0);

}
```

Compiled

```
fn main() {

    let opinion = Opinion("Systemd is bad".to_string());
    thing(opinion)

}

fn thing(s: Opinion) {

    println!("{}", s.0);

    drop(s); // added here by the rust compiler

}
```

Source

```
fn main() {

    let fact = Fact("I am not gaslighting you".to_string());
    thing(&fact)

}

fn thing(s: &Fact) {

    println!("{}", s.0);

}
```

Compiled

```
fn main() {

    let fact = Fact("I am not gaslighting you".to_string());
    thing(&fact)

    drop(fact);

}

fn thing(s: &Fact) {

    println!("{}", s.0);

}
```

Variables can only be owned once

```
fn main() {

    let opinion = Opinion ("Its Gif not Jif".to_string());

    speak(opinion);
    yell(opinion);

}
```

use of moved value: `opinion`

```
$ cargo build
   Compiling thing v0.1.0 (/home/tylerm/Documents/rust/thing)
error[E0382]: use of moved value: `opinion`
  --> src/main.rs:7:10
   |
5  |     let opinion = Opinion ("Its Gif not Jif".to_string());
   |         ------- move occurs because `opinion` has type `Opinion`, which does not implement the `Copy` trait
6  |     speak(opinion);
   |           ------- value moved here
7  |     yell(opinion);
   |          ^^^^^^^ value used here after move
   |
note: consider changing this parameter type in function `speak` to borrow instead if owning the value isn't necessary
  --> src/main.rs:11:14
   |
11 | fn speak(_s: Opinion) {
   |    -----     ^^^^^^^ this parameter takes ownership of the value
   |    |
   |    in this function
```

```rust
fn main() {

    let opinion = Opinion ("Its Gif not Jif".to_string());

    speak(opinion); // moved here
    yell(opinion); // opinion no longer in scope

}
```

- Borrow a value
- Move a value

Rust Ensures Memory Safety At Compile Time

- No Dangling References
- No Memory Leaks
- No Concurrency Errors
- More Efficient Memory Usage

Stack vs Heap

- Everything is put on to the stack (by default)
- Heap allocations are done by `Box::new()`

Fearless Concurrency

(🚀)

Rust Enforces Thread Safe Code

- No Mutable Static Variables
- Cross Thread Variables Must Be Locked

No Mutable Static Variables

- Static variables can be access from anywhere
- Multiple threads can access the state
- Concurrency issues

Arc

```
let counter = Arc::new(5);

let counter_two = Arc::clone(&counter);
some_func(counter_two)
```

Mutex

```
let counter = Mutex::new(5);

{
    let mut num = counter.lock().unwrap();
    *num = num + 1;
}

println!("{:?}", m);
```

```rust
let counter = Arc::new(Mutex::new(5));
let mut handles = vec![];

let thread_counter = Arc::clone(&counter);

thread::spawn(move || {

    let mut num = thread_counter.lock().unwrap();
    *num += 1;

}).join().unwrap();

println!("Result: {}", *counter.lock().unwrap());
```

lazy_static (crate)

```rust
lazy_static! {
    static ref NUM: Mutex<u64> = Mutex::new(0);
}

fn main() {
    let num = NUM.lock().unwrap();
    *num += 1;
}
```

There is no such thing as NULL in Rust

In other languages

- Return `null` instead of data
- Return `-1` for primitives
- Causes a lot of edge cases

Optional Results (C)

```c
struct RITStudent {
    int uid,
    bool is_broke
}

RITStudent getStudent(int id) {
    if (id > 0) {
        return /* The Student*/;
    } else {
        return NULL;
    }
}
```

Optional Results (Rust)

```
struct RITStudent {
    uid: u32,
    is_broke: bool
}

fn getStudent(id: u32) -> Option<RITStudent> {
    if (i > 0) {
        return Some(/* The Student */)
    } else {
        return None
    }
}
```

Returning Errors (Java)

```java
int assert_positive(int n) {
    if (n > 0) {
        return n;
    }
    throw new RuntimeException("haha program crash go brrrrrrr");
}

public static void main(String[] args) {
    int n = assert_positive(3);
}
```

Returning Errors (Rust)

```rust
fn is_positive(n: u32) -> Result<u32, String> {
    if (n > 0) {
        return Ok(n)
    } else {
        return Err("pls make n > 0 owo :3")
    }
}

fn main() {
    let n = is_positive(3);
}
```

Matching Options

```rust
fn main() {
    let option: Option<u8> = returns_option();
    match option {
        Some(n) => {
            // cool stuff with n
        },
        None => {
            // handle nothing
        }
    };
}
```

Matching Errors

```
fn main() {
    let option: Result<u8, String> = returns_error();
    match option {
        Some(n) => {
            // cool stuff with n
        },
        Err(err) => {
            // handle error
        }
    };
}
```

Other ways to handle options and results

```rust
fn main() {

    if let Err(e) = func() {
        // handle error
    }

    let Some(thing) = func2() else {
        // handle that you got nothing
    }

    if func2().is_none() {
        // handle that you got nothing
    }

}
```

Macros

```
#[derive(Debug)]
```

Macros

- Code that runs at compile time
- Generate new or modify existing code

function-like macros

Generates code in place of the macro

```rust
fn main() {
    println!("{}", 137);
    println!("{} {}", 137, "boe jiden");
}
```

attribute macros

Can be attached to `items` to generate or modify existing syntax

```
#[tokio:main]
fn main() {
    // tokio shit
}
```

derive macros

Can be attached to `structs` and add implementations to them

```rust
#[derive(Copy, Clone, Debug)]
pub struct Munson {
    compensation: u128
}
```

Different "Types" of Macros

- Declarative Macros
- Procedural Macros

Declarative Macros

- function-like
- Can only generate new code

```rust
macro_rules! show_result {
    ($expr:expr) => {
        println!("The result of '{}' is: {}", stringify!($expr), $expr)
    }
}

fn main() {
    show_result!(5 * 10 - 2);
}
// made by tristan :0
```

```
The result of '5 * 10 - 2' is: 48
```

Procedural Macros

- function-like, attribute, derive
- Can generate or modify existing syntax

```rust
#[proc_macro]
pub fn make_answer(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

```rust
make_answer!();

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

Tests

Tests are gud

Unit Testing is Built In

```rust
#[test]
fn test_the_thing() -> io::Result<()> {
    let state = setup_the_thing()?; // expected to succeed
    do_the_thing(&state)?;          // expected to succeed
    Ok(())
}

#[test]
#[should_panic(expected = "values don't match")]
fn mytest() {
    assert_eq!(1, 2, "values don't match");
}
```

Doc Tests

Makes sure documentation is always up to date0

```
/// ```
/// /// Some documentation.
/// # fn foo() {} // this function will be hidden
/// println!("Hello, World!");
/// ```
println!("Hello, World!");
```

Clippy (linter)

clippy my beloved 📎❤️

```
cargo clippy -- \
-W clippy::all \
-W clippy::nursery \
-W clippy::unwrap_used \
-W clippy::pedantic
```

This may seem like a lot

But its for a good reson

When you write code in rust

- The code will work
- Memory safe
- Thread safe
- Blazingly Fast
- It is easy to read

Now its time for a shitpost :)