
RUST PROGRAMMING
Expressions
Kamal kumar mukiri, Apt Computing Labs

Contents

Guess the output ???


```c
#include <stdio.h>

int add(int x, int y) {
}

int main() {
    int result = add(2, 3);
    printf("Result: %d\n", result);
    return 0;
}
```

ISSUES: Return value is undefined

Rust does not compile if there is no return value

Guess the output ???

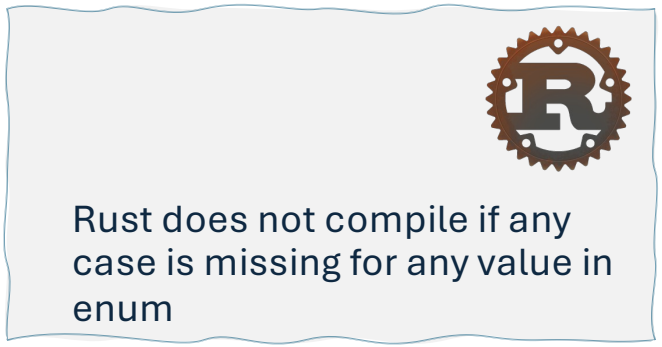
- Introduction
- Blocks
- Declarations
- If, let and else
- Types of Loops
- Functions
- Method calls
- Operators
- Assignments
- Typecasts
- Closures

```c
typedef enum COLORS {
    RED,
    GREEN,
    BLUE
} COLORS;

COLORS get_value(char *color) {
    if (strcmp(color, "red") == 0) {
        return RED;
    } else if (strcmp(color, "green") == 0) {
        return GREEN;
    } else {
        return 10;
    }
}

int main() {
    COLORS result = get_value("bule");
    switch (result) {
        case RED:
            printf("Red returned...");
            break;
        case GREEN:
            printf("Green returne...");
            break;
    }
    printf("Result: %d\n", result);
    return 0;
}
```

ISSUE: Handling return value is In-Complete



Rust does not leave even a single choice


```rust
#[derive(Debug)]
0 implementations
enum COLORS {
    RED,
    GREEN,
    BLUE,
    NOCOLOR
}

fn GetColorCode(name: &str) -> COLORS {
    let mut result: COLORS = COLORS::NOCOLOR;
    if name == "RED" {
        result = COLORS::RED;
    } else if name == "BLUE" {
        result = COLORS::BLUE;
    }
    return result;
}

fn main() {
    match GetColorCode(name: "BLUE") { // Switch
        COLORS::BLUE => println!("BLUE COLOR....."),
        COLORS::GREEN => println!("GREEN COLOR....."),
        _ => println!("Others here...") //Default
    }
}
```

- “match” -> Like a switch statement
- Programmer should write cases for all possible values

Statements

Syntax

Statement :

;
| *Item*
| *LetStatement*
| *ExpressionStatement*
| *MacroInvocationSemi*

Statements:

- Declaration statement
- Expression statement

Declaration Statements

Declaration statement

1. Item statement
2. ”let” statement

1

```
fn outer() {
  let outer_var = true;

  fn inner() { /* outer_var is not in scope here */ }

  inner();
}
```

2

Syntax

LetStatement :

*OuterAttribute** `let` *PatternNoTopAlt* (`:` *Type*)?

(`=` *Expression* † (`else` *BlockExpression*)?)? `;`

† When an `else` block is specified, the *Expression* must not be a *LazyBooleanExpression*, or end with a `}`.

“let” Statements

- Introduction
- Blocks
- Declarations
- If, let and else
- Types of Loops
- Functions
- Method calls
- Operators
- Assignments
- Typecasts
- Closures

refutable

[ri-**fyoo**-*tuh*-buhl, **ref**-*yuh*-*tuh*-] Phonetic (Standard) IPA

adjective

1 able to be proven false:

The statement is so vague as to be neither provable nor refutable.

1. **Refutable:** May fail to match a given value

```
let Some(forth) = b.pop() else { panic!();};
```

2. **Irrefutable**: Always match a given value.

```
let akhari = b.pop();
```

```
fn main() {
    let (mut a, mut b) = (String::from("Team"),
                          vec!["sachin",
                               "dravid",
                               "ganguly"]);
    a.push_str(" Cricket");


    // There is a chance that 'pop' may return None from Option<T>
1   let Some(last) = b.pop() else { panic!();};
    println!("last = {:?}", last);

2   let akhari = b.pop() else {panic!();};
    println!("Fifth = {:?}", akhari);
}
```

BTW what is Option?

Option<T> is an enum used to represent a value that may or may not be present.

The Option type forces you to handle the absence of a value explicitly.

Search for books from library with subject

Requirement:

- Program should read the data from ”json” file. Where key-value pair (Subject name (key) with all books (value)).
- Program should take argument from command line and display the list of books in library

Covers:

1) Reading command line argument
2) Using third party library (“crate” in Rust’s language)
3) Reading and writing from ”json” file
4) Using “HashMap”, a collector in Rust standard library
5) Writing an ”Enum”
6) Writing “match” expression
7) Discipline in Rust language while handling return value from function

Steps while starting a new project

1. **Understand Requirements**
2. **Evaluate Existing Libraries and Tools**
3. Set Up the Rust Project
    1. Create a New Project
    2. Set up Version Control
4. Planning Project Structure
    1. Modularize Your Code
    2. Organize Testing
    3. Handle Dependencies
5. Coding Best Practices
    1. Focus on Ownership and Borrowing
    2. Leverage Rust's Ecosystem
    3. Write Documentation
6. Test Early and Often
    1. Unit Tests
    2. Integration Tests
    3. Integration Tests
7. **Set Up Project Environment**
    1. Environment Variables
    2. Build Profiles
    3. Cross-Compilation
8. Document Your Code
    1. Write Documentation Comments
    2. Create a README
    3. Use cargo doc
9. Benchmarking and Optimization
    1. Use Criterion for Benchmarking
    2. Profiling
10. **Deploy and Package**
    1. Build for Release
    2. Distribute Binaries
11. Ongoing Maintenance
    1. Code Review and Refactoring
    2. Stay Updated

```
# Create a new Rust project
cargo new my_project --bin

# Move to the project directory
cd my_project

# Edit the Cargo.toml to add dependencies
nano Cargo.toml

# Write code in src/main.rs
nano src/main.rs

# Run the project
cargo run

# Test the project
cargo test

# Build for release
cargo build --release
```

Steps while starting a new project

1. **Understand Requirements**
   - Research the Problem Domain
   - Gather Requirements
   - Define Project Scope

2. **Evaluate Existing Libraries and Tools**
   - Search for Existing Solutions
   - Check Documentation
   - Consider Performance Requirements

3. **Set Up the Rust Project**
   - Install rust
   - Create a New Project
   - Set up Version Control

4. **Planning Project Structure**
   - Modularize Your Code
   - Organize Testing
   - Handle Dependencies

5. **Coding Best Practices**
   - Focus on Ownership and Borrow
   - Leverage Rust's Ecosystem

6. **Document Your Code**
   - Write Documentation Comments
   - Create a README
   - Use cargo doc

7. **Test Early and Often**
   - Unit Tests
   - Integration Tests

8. **Set Up Project Environment**
   - Environment Variables
   - Build Profiles
   - Cross-Compilation

9. **Benchmarking and Optimization**
   - Use Criterion for Benchmarking
   - Profiling

10. **Deploy and Package**
    - Build for Release
    - Distribute Binaries

11. **Ongoing Maintenance**
    - Code Review and Refactoring
    - Stay Updated

Expressions

Expresssion Type	Example	Descrption
Array Literal	`let array = [1,2,3];`	
Repeat Array Literal	`let array = [0;50]`	It creates array with size 50 and assigns all values with 0
Tuple	`let msg = (101, "SSH password")`	Multiple data types are supported
Grouping	`(2+2)`	
Block	`{ f(); g() }`	Return value of g() is the value of **block**.

Control flow expressions	`if ok { f() }`
	`if ok { 1 } else { 0 }`
	`if let Some(val) = f() { x } else {y}`
	`match x {None=>0, _ => 1}`
	`for v in e { f(v); }`
	`while ok { ok = f(); }`
	`while let Some(val) = it.next() {f(x);}`
	`loop ( next_event(); }`
	`break;`
	`continue;`
	`return 0;`	

Macro Invocation	`println!("ok");`	
path	`std::f64::consts::PI;`	

Expressions

Expresssion Type	Example	Descrption
path	`std::f64::consts::PI;`	
Structural litertal	`let p1 = Point {x:0, y:0};` `Accessing: p1.x`	
Fucntion Call	`stdin()`	Traits: Kinds of closures (anonymous functions) Fn(Arg0, ...) -> T FnMut(Arg0, ...) -> T FnOnce(Arg0, ...) -> T
Index	`let a = [1,2,3];` `printlnn!("a[0] = ", a[0]);`	
Error Check	`creatdir("./name")?;`	
Logocal bitwise	`!ok`	Not
Negation	`-num`	Neg
Dereference	`*ref`	Deref, DerefMut
Typecast	`x as i32`	
Arithmetic	`+, -, *, /, %`	Add, Sub, Mul, Div, Rem
Bitwise	`<<, >>, &, ^, |`	Shl, Shr, BitAnd, BitXor, BitOr
Comparison	`<, >, <=, +>, ==, !=`	std::cmp::PartialOrd std::cmp::PartialEq
Logical	`&&, ||`	
End-Exclusive Range	`start..stop`	
End-Inclusive Range	`start..=stop`	

Anonymous functions


```
let closure = |parameters| expression;
```

Types:

1. **Simple closures**
2. **Closures capturing environments**
3. **Closures with explicit types**
4. **Closures returning value**

Characteristics of Rust Closures:

1. **Capture Environment**: Closures can capture variables from their surrounding environment, which is different from regular functions.
2. **Type Inference**: Rust can usually infer the types of closure parameters.
3. **Can Return Closures**: Functions in Rust can return closures using the impl Fn, impl FnMut, or impl FnOnce syntax.

Simple Closures


```
fn main() {
    let add = |a, b| a + b;
    let add2 = |a, b| a + b;
    let result = add(2, 3); // Using the closure
    let result2 = add2(2.1, 3.1);
    println!("2 + 3 = {}", result);
    println!("2.1 + 3.1 = {}", result2);
}
```

Output

```
2 + 3 = 5
2.1 + 3.1 = 5.2
```

1. Simple closure (add)

Closure capturing environment


```
fn main() {
    let tax_perc = 0.3; //30%
    let calculate_tax = |salary| println!("tax = {}",
    salary*tax_perc);
    calculate_tax(100000.0);
}
```

1. Taking tax_perc as environment variable

Output

```
> ./Types_closures
tax = 30000
```

Closure with explicit types


```
fn main() {
    let tax_perc = 0.3; //30%
    let calculate_tax = |salary:f64| println!("tax =
    {}", salary*tax_perc);
    calculate_tax(100000 as f64);
}
```

Output

```
> ./Types_closures
tax = 30000
```

1. Accepts only “f64”

Closure returning a value


```
fn main() {
    let calculate_square = |num: i32| -> i32 {
    num * num
    };

    let result = calculate_square(5); // Closure
    returns the square of 5
    println!("Square of 5 is: {}", result);
}
```

Output

```
> rustc Types_closures.rs
> ./Types_closures
Square of 5 is: 25
```

1. Retutype is “i32”

Thank you