

ORGANIZER:



GFI



VBI

POWERED BY:



WEB3HACKFEST

DEV PARTNER:

CODE
mely

VU NGUYEN
{ C <> I } E R }

RUST DEVELOPER BOOTCAMP

📅 19:30 - 21:00 | 03/07/2023

🎧 Discord, Zoom Online





Struct, Enum, Vector, Generic Type

1. Struct



Struct: Tập hợp các kiểu dữ liệu khác nhau

```
fn main() {
```

```
    let student_a = Student { name: "John".to_string(), age: 20, class: "B1".to_string() }
```

```
}
```

```
struct Student {
```

```
    name: String,
```

```
    age: u8,
```

```
    class: String, }
```

struct
Student
name, age, class
student_a

1. Struct : Mô tả hành vi



```
impl Student {  
    fn get_name(self) -> String {  
        self.name  
    }  
    fn print_name(self) {  
        println!("Name: {}", self.get_name());  
    }  
}
```

Sử dụng từ khoá impl

1. Struct: Vấn đề 1 nếu không có từ khoá **self**

```
impl Student{  
    fn get_name()-> String{  
        name  
    }  
}
```

error[E0425]: cannot find value `name` in this scope
--> src/main.rs:19:9

```
19 |         name  
    |         ^^^^ a field by this name exists in `Self`
```

1. Struct: Vấn đề 2 từ khoá **Self**

```
impl Student{
  fn new()-> Student{
    Student{
      name:String::from("Mike"),
      age:24,
      class:String::from("B"),
    }
  }
}
```

```
impl Student{
  fn new()-> Self{
    Self{
      name:String::from("Mike"),
      age:24,
      class:String::from("B"),
    }
  }
}
```

1. Struct: dùng self, &self, &mut self

```
struct Object{  
    width: i32,  
    length: i32,  
}  
  
impl Object{  
    fn new(width: i32, length: i32)-> Object {  
        Object {width: width,length: length}  
    }  
    fn area(self)-> i32 {  
        self.width*self.length  
    }  
}
```

```
let p= Object {  
    width: 50,  
    length: 50,  
};  
  
println!("{}", p.width, p.length,p.area());
```

self: Ownership

1. Struct: dùng self, &self, &mut self

```
fn area(self) -> i32 {  
    self.width * self.length  
}
```

```
let p = Object {  
    width: 50,  
    length: 50,  
};
```

```
println!("{}", p.width, p.length, p.area());  
println!("{}", p.width, p.length, p.area());
```

```
println!("{}", p.width, p.length, p.area());
```

-----^-----
| | |
| | | move out of `p` occurs here
| | borrow of `p.width` occurs here
| borrow later used here

Lỗi

1. Struct: dùng self, &self, &mut self

```
fn area(&self) -> i32 {  
    self.width * self.length  
}
```

```
let p = Object {  
    width: 50,  
    length: 50,  
};  
  
println!("{}", p.width, p.length, p.area());  
println!("{}", p.width, p.length, p.area());
```

&self: shared Reference

1. Struct: dùng self, &self, &mut self

```
fn increase(&mut self) {  
    self.width = self.width + 20;  
    self.length = self.length + 20;  
}
```

```
let mut p = Object {  
    width: 50,  
    length: 50,  
};  
println!("{}", p.width, p.length, p.area());  
p.increase();  
println!("{}", p.width, p.length, p.area());
```

&mut self: Mutable Reference

2. Enum



```
#[derive(Debug)]
enum Position{
    One,
    Two,
    Three
}
```

```
#[derive(Debug)]
enum Person {
    Peter(Position),
    Adam(Position)
}
```

```
let one = Position::One;
let two = Position::Two;
let who = Person::Peter(Position::One);
println!("{:?}", one);
println!("{:?}", who);
```

2. Enum

```
#[derive(Debug)]  
enum Position{  
    One,  
    Two,  
    Three  
}
```

```
#[derive(Debug)]  
enum Person {  
    Peter(Position),  
    Adam(Position)  
}
```

```
let one = Position::One;  
let two = Position::Two;  
let who = Person::Peter(Position::One);  
println!("{:?}", one);  
println!("{:?}", who);
```

2. Enum

```
#[derive(Debug)]  
enum Info{  
    Peter(Student),  
    Adam(Student)  
}
```

```
let student_a = Student{  
    name:String::from("John"),  
    age:20,  
    class:String::from("A"),  
};  
let info = Info::Peter(student_a);  
println!("{:?}", info);
```

2. Enum



```
enum Direction { North, East, South, West }
```

```
fn main() {  
    let direction:Direction = Direction::North;  
    match direction {  
        Direction::North => {  
            println!("Direction is north");  
        },  
        Direction::East => {  
            println!("Direction is East");  
        },  
        Direction::South => {  
            println!("Direction is South");  
        },  
        Direction::West => {  
            println!("Direction is West");  
        }  
    }  
}
```

3. Vec



`let mut a = Vec::new();` //1. Sử dụng `new()` method

`let mut b = vec![];` //2. Sử dụng `vec!` macro

3. Vec



//Lấy giá trị và thay đổi giá trị

```
let mut c = vec![5, 4, 3, 2, 1];
```

```
c[0] = 1;
```

```
c[1] = 2;
```


3. Vec



//push and pop

```
let mut d: Vec<i32> = Vec::new();
```

d.push(1); //[1] : Thêm giá trị vào vị trí cuối cùng của vec

d.push(2); //[1, 2]

d.pop(); //[1] : : Xoá giá trị vào vị trí cuối cùng của Vec

3. Vec



```
let mut v = vec![1, 2, 3, 4, 5];
```

```
for i in &v {
```

```
    println!("A reference to {}", i);
```

```
}
```

```
for i in &mut v {
```

```
    println!("A mutable reference to {}", i);
```

```
}
```

```
for i in v {
```

```
    println!("Take ownership of the vector and its  
element {}", i);
```

```
}
```

4. Generic type


Generic type là kiểu dữ liệu chung (placeholder) có thể thay thế cho các kiểu dữ liệu Rust

```
fn main() {  
    let x = get_u8(10u8);  
    let y = get_u8(10u16);  
}  
  
fn get_u8(input: u8) -> u8 {  
    input  
}
```

Lỗi

4. Generic type

```
fn main() {  
  
    let x = get_u8(10u8);  
    let y = get_u8(10u16);  
}  
  
fn get_u8<T>(input: T) -> T{  
    input  
}
```



```
fn get_u8(input: u8) -> u8{  
    input  
}
```

```
fn get_u8(input: u16) -> u16{  
    input  
}
```



4. Generic type in Struct

```
impl<T> Point<T> {  
    fn get_x(&self) -> &T {  
        &self.x  
    }  
}
```

```
let integer = Point { x: 5, y: 10 };  
let float = Point { x: 1.0, y: 4.0 };  
println!("integer.x = {}", integer.get_x());  
println!("float.x = {}", float.get_x());
```



4. Generic type in Struct

```
struct Point<T> {  
    x: T,  
    y: T,  
}
```

```
fn main() {  
  
    let integer = Point { x: 5, y: 10 };  
    let float = Point { x: 1.0, y: 4.0 };  
  
}
```

4. Generic type in Struct

```
struct Point<T, U> {  
    x: T,  
    y: U,  
}
```

```
let integer = Point { x: 5, y: 10.5 };  
let float = Point { x: 1.5, y: 4.0 };  
println!("integer.x = {}", integer.get_x());  
println!("float.x = {}", float.get_x());
```

```
impl<T,U> Point<T,U> {  
    fn get_x(&self) -> &T {  
        &self.x  
    }  
}
```

4. Generic type in Enum



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


4. Generic type in Enum



Option

Làm sao lấy giá trị trong Option?

```
fn main() {  
  let x: Option<i32> = Some(5);  
  let y: Option<f64> = Some(5.0f64);  
}
```

5. Lifetime



- + Lifetime nghĩa là thời gian tồn tại của biến diễn ra trong bao lâu (dựa vào scope và tính ownership)
- + Thường diễn ra đối với reference

5. Lifetime

Reference không thể tồn tại
lâu hơn so với object mà ta đã
`mượn`

```
fn returns_reference() -> &str {  
    let my_string = String::from("I am a string");  
    &my_string // ⚠  
}
```

```
fn main() {}
```

5. Lifetime



```
#[derive(Debug)]
struct City {
    name: &str, // ⚠️
    date_founded: u32,
}

fn main() {
    let my_city = City {
        name: "Ichinomiya",
        date_founded: 1921,
    };
}
```

5. Lifetime



Một số cách :

- + Sử dụng owned type (clone)
- + Sử dụng cách đánh lifetime

6. Thực hành





Cảm ơn mọi người đã lắng nghe