GFS BLOCKCHAIN VBI VN BLOCKCHAIN INNOVATION VIETNAM BLOCKCHAIN UNION octopus network

MIỄN PHÍ KHÓA ĐÀO TẠO

LẬP TRÌNH BLOCKCHAIN TÙY CHỈNH TRÊN SUBSTRATE OCT + MINIHACKATHON

Nội Dung Khóa Học:

Phần 1:
Làm quen với lập trình rust cơ bản (2-3 tuần)

Phần 2:
Làm quen cơ bản với substrate theo hướng dẫn (1 Tuần)

Phần 3:
Lập trình Blockchain nâng cao (thực chiến với giảng viên 6 tuần)

Phần 4:
Teamup tham gia Minihackathon (2 tuần)

Hạn chót đăng ký: 07/06/2022

Giải Thưởng: 4.000$ /khóa

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

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

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

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

```rust
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}

}
```

```rust
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

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

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

2. Enum

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

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

2. Enum

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

```rust
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);

}
```

3. Vec

Phân biệt iter(), into_iter(), iter_mut(),

The iterator returned by into_iter may yield any of T, &T or &mut T, depending on the context.

The iterator returned by iter will yield &T, by convention.

The iterator returned by iter_mut will yield &mut T, by convention.

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

```rust
fn main() {

    let x = get_u8(10u8);
    let y = get_u8(10u16);
}

fn get_u8<T>(input: T) -> T{
    input
}
```

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

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

4. Generic type in Struct

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

```rust
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

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

```rust
fn main() {

    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };

}
```

4. Generic type in Struct

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

```rust
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());
```

```rust
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. Thực hành

Bài 1: Điền vào dấu chấm hỏi

```
fn main() {
    let mut shopping_list: Vec<?> = Vec::new();
    shopping_list.push("milk");
}
```

5. Thực hành

Bài 2: Trường dữ liệu value của Wrapper có thể sử dụng u32 hoặc String hoặc …

```
struct Wrapper {
    value: u32,
}
impl Wrapper {
    pub fn new(value: u32) -> Self {
        Wrapper { value }
    }
}
```

5. Thực hành

Bài 3: Bước 1: Thực hiện một implement in ra tất cả các trường dữ liệu. Bước 2: đối với trường dữ liệu **grade**, ta có thể có trường hợp là “A+”, “B+” ,...

```
pub struct ReportCard {

    pub grade: f64,

    pub student_name: String,

    pub student_age: u8,

}
```

5. Thực hành

Bài 4: TODO

```
enum Message {
    // TODO: define a few types of messages as used below
}
fn main() {
    println!("{:?}", Message::Quit);
    println!("{:?}", Message::Echo);
    println!("{:?}", Message::Move);
    println!("{:?}", Message::ChangeColor);
}
```

5. Thực hành

```
#[derive(Debug)]
enum Message {
    // TODO: define the different variants
 used below
}

impl Message {
    fn call(&self) {
        println!("{:?}", &self);
    }
}
```

```
fn main() {
    let messages = [
        Message::Move { x: 10, y: 30 },
        Message::Echo(String::from("hello world")),
        Message::ChangeColor(200, 255, 255),
        Message::Quit,
    ];

    for message in &messages {
        message.call();
    }
}
```

5. Thực hành

```
fn print_number(maybe_number: Option<u16>) {
    println!("printing: {}", maybe_number.unwrap());
}

fn main() {
    print_number(13);
    print_number(99);

    let mut numbers: [Option<u16>; 5];
    for iter in 0..5 {
        let number_to_add: u16 = {
            ((iter * 1235) + 2) / (4 * 16)
        };

        numbers[iter as usize] = number_to_add;
    }
}
```

5. Thực hành

```
#[derive(Debug, Clone)]
struct MyData {
    val1: i32,
    val2: String,
}
    fn main() {
        let d = MyData {
            val1: 35,
            val2: String::from("Hello World"),
        };

        let both = d.get_both();
        let x = d.get_val1();
        let y = d.get_val2();
    }
```

```
impl MyData {
    pub fn get_val1(self) -> i32 {
        return self.val1.clone();
    }

    pub fn get_val2(self) -> String {
        return self.val2.clone();
    }

    pub fn get_both(self) -> (i32, String)
        return (self.val1, self.val2);
    }
}
```

5. Thực hành

```
fn main() {
    let a = A {p: Some("p".to_string())};
    a.a();
}

struct A {
    p: Option<String>
}
```

```
impl A {
    fn a(self) -> Self {
        Self::b(&self.p.unwrap());
        self
    }
    fn b(b: &str) {
        print!("b: {}", b)
    }
}
```

5. Thực hành

```
// Fill in the blanks to make it work
struct A;          // Concrete type `A`.
struct S(A);       // Concrete type `S`.
struct SGen<T>(T); // Generic type `SGen`.

fn reg_fn(_s: S) {}

fn gen_spec_t(_s: SGen<A>) {}

fn gen_spec_i32(_s: SGen<i32>) {}

fn generic<T>(_s: SGen<T>) {}

fn main() {
    // Using the non-generic functions
    reg_fn(__);          // Concrete type.
    gen_spec_t(__);   // Implicitly specified type parameter `A`.
    gen_spec_i32(__); // Implicitly specified type parameter `i32`.

    // Explicitly specified type parameter `char` to `generic()`.
```

5. Thực hành

```
// Implement the generic function below.
fn sum

fn main() {
    assert_eq!(5, sum(2i8, 3i8));
    assert_eq!(50, sum(20, 30));
    assert_eq!(2.46, sum(1.23, 1.23));

    println!("Success!");
}
```

5. Thực hành

```
// Implement struct Point to make it work.


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

    println!("Success!");
}
```

5. Thực hành

```
// Modify this struct to make the code work
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    // DON'T modify this code.
    let p = Point{x: 5, y : "hello".to_string()};

    println!("Success!");
}
```

5. Thực hành

```
// Add generic for Val to make the code work, DON'T modify the code
in `main`.
struct Val {
    val: f64,
}

impl Val {
    fn value(&self) -> &f64 {
        &self.val
    }
}


fn main() {
```

5. Thực hành

```
// Fix the errors to make the code work.
struct Point<T> {
    x: T,
    y: T,
}

impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```