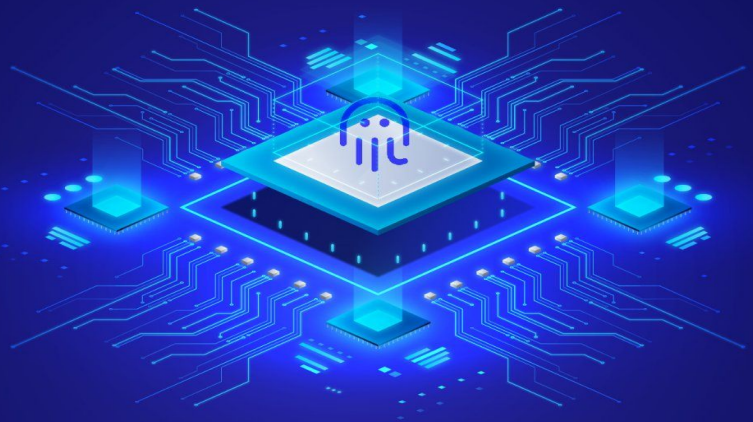
GFS BLOCKCHAIN · VBI · VIETNAM BLOCKCHAIN UNION · octopus network

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

Trait and Lifetime

Bài tập struct

```
#![allow(dead_code)]


struct Store {
    name: String,
    items: Vec<Item>,
}


#[derive(Debug)]
struct Item {
    name: String,
    price: f32,
}


impl Store {
    fn new(name: String) -> Store {
        Store {
            name: name,
            items: vec![],
```

Lifetime in Rust

+ Every reference in Rust has a lifetime
+ Trong một số trường hợp ta cần biến reference có vòng đời (lifetime) không phụ thuộc vào scope
+ Ngăn chặn trường hợp “dangling reference"
+ Borrow checker

Cách sử dụng:

Sử dụng generic parameter

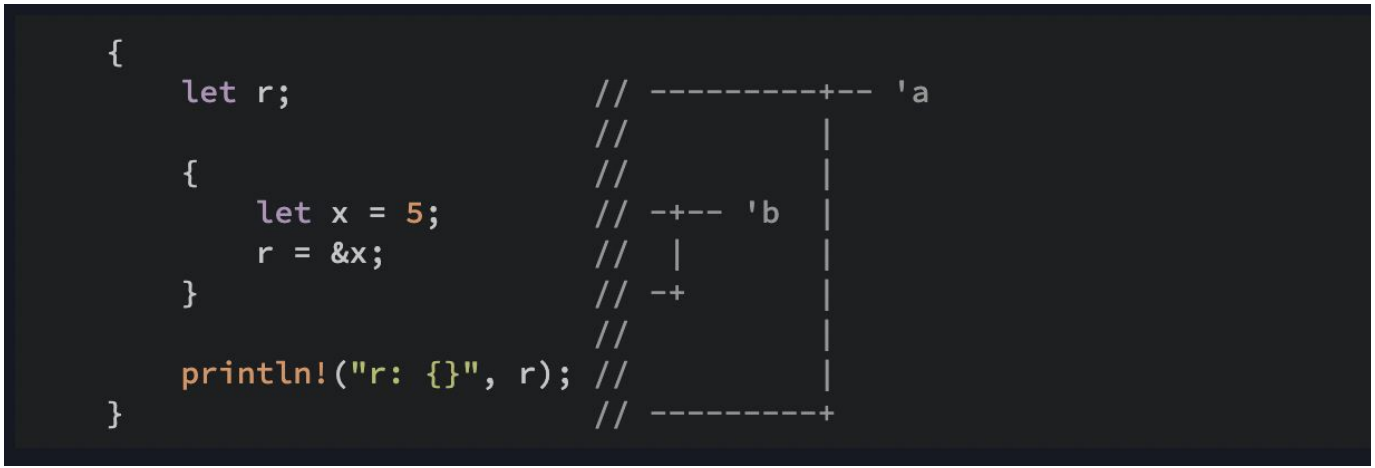
Lifetime in Rust

```rust
fn main() {
    {
        let r;

        {
            let x = 5;
            r = &x;
        }

        println!("r: {}", r);
    }
}
```

```
error[E0597]: `x` does not live long enough
 --> src/main.rs:5:12
  |
5 |         r =&x;
  |            ^^ borrowed value does not live long enough
6 |
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r:{}",r);
  |                     - borrow later used here
```

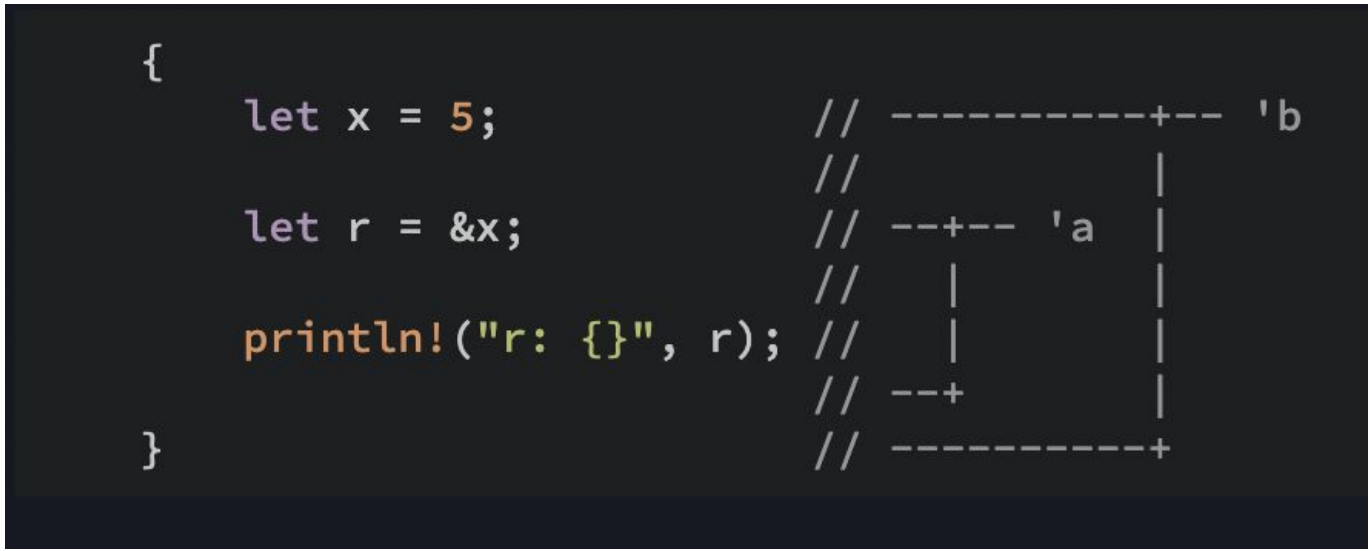
Lifetime in Rust

Borrow checker sẽ kiểm tra biến borrowing đang ở lifetime nào so với biến mới nhất owner



Lifetime in Rust

Cách giải quyết



Lifetime in Rust

Cú pháp để đánh dấu lifetime

```
&i32        // a reference

&'a i32     // a reference with an explicit lifetime

&'a mut i32 // a mutable reference with an explicit lifetime
```

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1, string2.to_string());
    println!("The longest string is {}", result);
}
```

```
fn longest(x: String, y: String) -> String {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime in Rust

Đánh dấu lifetime

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Trait

Trait là gì?

Định nghĩa các hành vi chia sẻ (shared behaviour) của một kiểu dữ liệu chưa biết trong Rust / interface

Mục đích của Trait

+ shared behaviour
+ Code reuse

Trait

Cách định nghĩa Trait

```
trait Speak {
    fn say_hello(&self) -> String;
}
```

Cách sử dụng Trait

```
struct Person {}
impl Speak for Person {
    fn say_hello(&self) -> String {
        String::from("Hello!")
    }
}
```

Hàm main

```
fn main(){
    let person1 = Person{};
    let res = person1.say_hello();
    println!("say something: {}", res);
}
```

Trait Default Implementation

Implement mặc định trong Trait

```
trait Speak {
    fn say_hello(&self) -> String {
        String::from("Hello!")
    }
}
```

Trường hợp ko sử dụng Implement mặc định

```
struct Person3 {}
impl Speak for Person3 {
    fn say_hello(&self) -> String {
        String::from("Hello World!")
    }
}
```

Trường hợp kiểu dữ liệu sử dụng implement mặc định

```
struct Person1 {}
impl Speak for Person1 {}
struct Person2 {}
impl Speak for Person2 {}
```

Trait in function arguments and trait bound

```
fn give_greeting(p: impl Speak) {
    println!("{}", p.say_hello());
}
```

Tại sao sử dụng trait như là 1 parameter?

Định nghĩa theo kiểu Trait Bound

```
fn give_greeting<T: Speak>(p: T) {
    println!("{}", p.say_hello());
}
```

```
fn give_greeting<T>(p: T)
    where T: Speak
{
    println!("{}", p.say_hello());
}
```

Trait in function arguments and Associated Type

```rust
pub trait Iterator<T> {
    fn next(&mut self) -> T;
}
```

Generic type

```rust
struct Counter{
    x:u32
}
impl Iterator<u32> for Counter{

    fn next(&mut self) -> u32 {
        self.x = self.x +1;
        self.x
    }
}
```

Trait in function arguments and Associated Type

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Self::Item;
}
```

Associated type

```rust
struct Counter{
    x:u32
}
impl Iterator for Counter{
    type Item =u32;
    fn next(&mut self) -> Self::Item{
        self.x = self.x +1;
        self.x
    }
}
```

```rust
let mut count = Counter{x:1};
println!("next_is:{}", count.next());
println!("next_is:{}", count.next());
```

Trait in function arguments and Associated Type

Vấn đề khi sử dụng Generic Type

```
struct Container(i32, i32);

trait Contains<A, B> {
    fn contains(&self, _: &A, _: &B) -> bool;
    fn first(&self) -> i32;
    fn last(&self) -> i32;
}
```

```
impl Contains<i32, i32> for Container {
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }

    fn first(&self) -> i32 { self.0 }

    fn last(&self) -> i32 { self.1 }
}
```

Trait in function arguments and Associated Type

Vấn đề khi sử dụng Generic Type

```
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> {
    container.last() - container.first()
}
```

Nhiều generic type quá -> phức tạp

Associated Type

```
trait Contains {
    type A;
    type B;
    fn contains(&self, &Self::A, &Self::B) -> bool;
}
```

Trait in function arguments and Associated Type

Khi implement Trait cho 1 Kiểu dữ liệu nào đó

```
impl Contains for Container {

    type A = i32;
    type B = i32;
```

```
fn difference<C: Contains>(container: &C) -> i32 {
    container.last() - container.first()
}
```

Trait in function arguments and Associated Type

Associated Type	Generic type
Cũng là 1 generic type	
Rút gọn code (dễ đọc code) khi có nhiều generic type	Vấn đề khi có quá nhiều generic type trong 1 trait hoặc 1 struct hoặc 1 function, …
Có thể định nghĩa các loại associated type cần dùng, không nhất thiết phải sử dụng hết	Bắt buộc sử dụng các generic type khi mà định nghĩa

Trait combos

Có thể sử dụng nhiều trait cùng 1 lúc

T: Trait1 + Trait2 + Trait3

Generic Parameter & Trait object

```
trait Processor {
    fn compute(&self, x: i64, y: i64) -> i64;
}

struct Risc {}

impl Processor for Risc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x + y
    }
}

struct Cisc {}

impl Processor for Cisc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x * y
    }
}

fn process<P: Processor>(processor: &P, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}

pub fn main() {
    let processor1 = Cisc {};
    let processor2 = Risc {};

    process(&processor1, 1);
    process(&processor2, 2);
}
```

Generic Parameter & Trait object

```
trait Processor {
    fn compute(&self, x: i64, y: i64) -> i64;
}

struct Risc {}

impl Processor for Risc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x + y
    }
}

struct Cisc {}

impl Processor for Cisc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x * y
    }
}

fn process(processor: &dyn Processor, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}

pub fn main() {
    let processors: Vec<Box<dyn Processor>> = vec![
        Box::new(Cisc {}),
        Box::new(Risc {}),
    ];

    for processor in processors {
        process(&*processor, 31);
    }
}
```

Static Dispatch & Dynamic Dispatch

#[derive()]

Compiler cung cấp cho developer 1 số basic implementation cho 1 số trait thông qua macro **derive**

+ So sánh: PartialEq
+ Tạo T từ &T : Clone
+ Tạo instance mặc định: Default
+ Sử dụng format gía trị {:?}

Thực hành

```
use std::io;
fn main() {
    let mut input: Vec<&str>;
    loop {
        let mut input_text = String::new();
        println!("Type instruction in the format Add <name> to <department>:");
        io::stdin().read_line(&mut input_text).expect("failed to read from stdin");
        let trimmed_text: String = input_text.trim().to_string();
        input = trimmed_text.split(" ").collect();
        if input[0] == "Add" && input[2] == "to" {
            break;
        } else {
            println!("Invalid format.");
        }
    }
    println!("{:?}", input);
}
```

Thực hành

```
trait AppendBar {
    fn append_bar(self) -> Self;
}

impl AppendBar for String {
    //Add your code here
}

fn main() {
    let s = String::from("Foo");
    let s = s.append_bar();
    println!("s: {}", s);
}
```

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn is_foo_bar() {
        assert_eq!(String::from("Foo").append_bar(), String::from("F
    }

    #[test]
    fn is_bar_bar() {
        assert_eq!(
            String::from("").append_bar().append_bar(),
            String::from("BarBar")
        );
    }
}
```

Thực hành

```
trait AppendBar {
    fn append_bar(self) -> Self;
}
//TODO: Add your code here
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn is_vec_pop_eq_bar() {
        let mut foo = vec![String::from("Foo")].append_bar();
        assert_eq!(foo.pop().unwrap(), String::from("Bar"));
        assert_eq!(foo.pop().unwrap(), String::from("Foo"));
    }
}
```

```rust
#![allow(dead_code)]

////////////////////////////////////////////////////////////////////////////////
// The trait
////////////////////////////////////////////////////////////////////////////////

trait Price {
    fn price(&self, item_name: &str) -> Option<f32>;
}

////////////////////////////////////////////////////////////////////////////////
// Store
////////////////////////////////////////////////////////////////////////////////

struct Store {
    name: String,
    items: Vec<Item>,
}

#[derive(Debug)]
struct Item {
    name: &'static str,
    price: f32,
}

impl Store {
    fn new(name: String) -> Store {
        Store {
            name: name,
            items: Vec::new(),
        }
    }

    fn add_item(&mut self, item: Item) {
        self.items.push(item);
    }
}

impl Price for Store {
    fn price(&self, item_name: &str) -> Option<f32> {
        for item in &self.items {
            if item.name == item_name {
                return Some(item.price);
            }
        }
        None
    }
}

fn build_store() -> Store {
    let mut store = Store::new(format!("Rustmart"));
    store.add_item(Item { name: "chocolate", price: 5.0 });
    store.add_item(Item { name: "socks", price: 23.0 });
    store.add_item(Item { name: "plush Mozilla dinosaur", price: 13.0 });
    store
}

////////////////////////////////////////////////////////////////////////////////
// Factory
////////////////////////////////////////////////////////////////////////////////

// A factory for just a single kind of item
struct Factory {
    item_name: &'static str,
    wholesale_price: f32,
}

impl Price for Factory {
    fn price(&self, item_name: &str) -> Option<f32> {
        // Goal: return the price of the factory's item, if the name matches;
        // otherwise return `None`.
        None
    }
}

fn build_factory() -> Factory {
    Factory {
        item_name: "sprocket",
        wholesale_price: 7.67,
    }
}

////////////////////////////////////////////////////////////////////////////////
// Total price
////////////////////////////////////////////////////////////////////////////////

fn total_price<P: Price>(provider: &P, shopping_list: &[&str]) -> Option<f32> {
    // Goal: compute the total price of all items in the shopping
```

```rust
#![allow(dead_code)]

////////////////////////////////////////////////////////////////////////////

// The trait

////////////////////////////////////////////////////////////////////////////

trait Price {
    fn price(&self, item_name: &str) -> Option<f32>;

    fn total_price(&self, shopping_list: &[&str]) -> Option<f32> {
        // Goal: compute the total price of all items in the shopping
        // list. If any of the options are not present, return `None`.
        None
    }
}

////////////////////////////////////////////////////////////////////////////

// Store
```