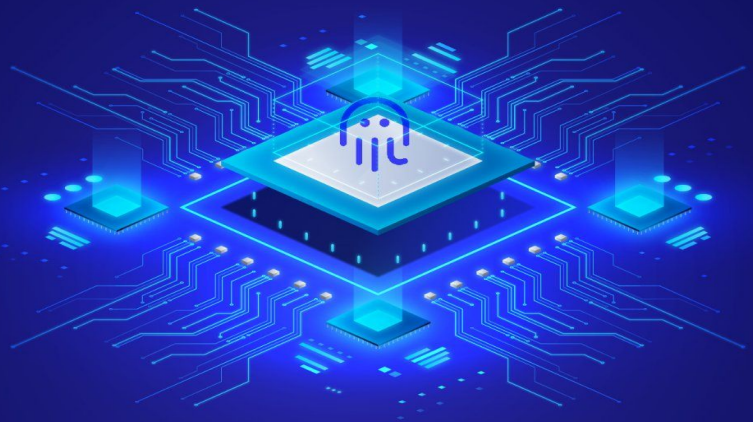
GFS BLOCKCHAIN · VBI · VIETNAM BLOCKCHAIN UNION · octopus network

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

Class 2: Ownership and Borrowing

1. Function

+ Tập hợp các đoạn code để thực hiện một logic, nhiệm vụ nào đó
+ Các lệnh code sẽ thực hiện từ trên xuống dưới
+ Hàm có tham số, hàm không có tham số, hàm có giá trị trả về, hàm không trả về (void), hàm lồng hàm

Tại vì sao phải sử dụng Function:

+ Chức năng lớn -> chia ra các chức năng nhỏ hơn -> dễ kiểm soát logic
+ Có thể sử dụng lại cho các mục đích khác nhau
+ Thường các thư viện sẽ định nghĩa các function -> 1 mục đích nhất định -> developer có thể sử dụng (https://doc.rust-lang.org/std/index.html)

1. Function

```rust
fn function_with_param(x: u32, y: u32) -> u32{
    x+y
}
```

```rust
fn function_without_param() {
    println!("Hello World");
}
```

1. Function

Một số lưu ý khi sử dụng hàm

+ Tham số hàm định nghĩa kiểu dữ liệu gì, thì khi sử dụng lại hàm đó phải đúng kiểu dữ liệu của tham số
+ Đối với hàm có trả về, thì kết thúc hàm phải trả về đúng kiểu dữ liệu mà hàm đã định nghĩa
+ Có 2 cách để trả giá trị hàm : **biểu thức hoặc biến ko có dấu chấm phẩy** ở đoạn cuối của hàm hoặc dùng từ khoá **return**

2. Ownership

Có 2 trường hợp cụ thể như sau:

```
// let x = 5;
// let y =x;
// println!("x:{}",x);
// println!("y:{}",y);
```

A

```
// let s1 = String::from("hello");
// let s2 = s1;
// println!("s1: {}", s1);
// println!("s2: {}", s2);
```

B

2. Ownership

Để hiểu giải thích vấn đề trên cần tìm hiểu 2 vấn đề nữa

+ Phân biệt giá trị, biến , pointer
+ Stack/Heap

2. Ownership

Giá trị: là sự kết hợp giữa kiểu dữ liệu và một phần tử trong số miền giá trị (i8 =>
- 2^7 ->2^7-1)

Biến : Đại diện cho giá trị , lưu giá trị ở một vùng nhớ nào đó

Pointer: là giá trị nhưng giữ địa chỉ trỏ tới vùng nhớ mà giá trị được lưu

2. Ownership

let x = 42;

let y = 43;

let var1 = &x;

let mut var2 = &x;

var2 = &y

2. Ownership

Stack

```
fn foo() {
    let y: i32 = 5;
    let z: i32 = 100;
}

fn main() {
    let x: i32 = 42;

    foo();
}
```

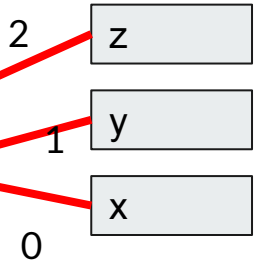
Address	Name	Value
2	z	100
1	y	5
0	x	42

Khi hết scope của hàm foo()

Address	Name	Value
0	x	42

Khi hết scope của hàm main()

Stack rỗng



2. Ownership

Heap

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

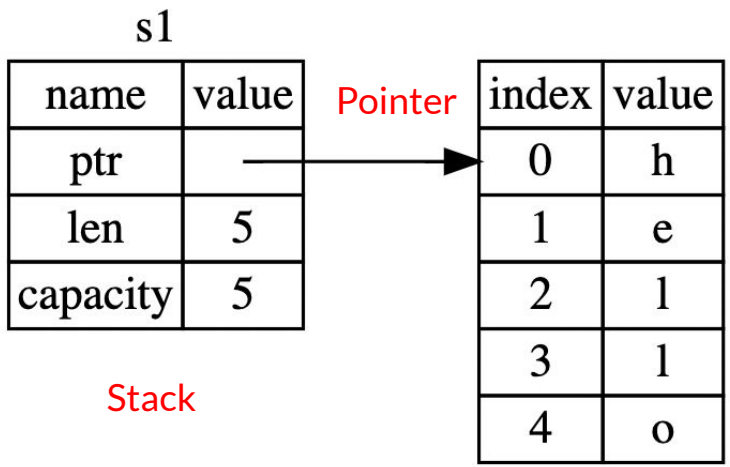
https://doc.rust-lang.org/stable/std/boxed/struct.Box.html

2. Ownership

Stack	Heap
Việc tạo vùng nhớ cho giá trị, việc đọc giá trị nhanh	Runtime perfomance cost
Vùng nhớ sẽ tự động xoá sau khi biến thoát ra khỏi scope	Vùng nhớ sẽ tự động xoá khi owner cuối cùng thoát ra khỏi scope Vùng nhớ có thể “live" ngoài cái phạm vi mà đã tạo ra nó
Set mặc định trong Rust	
Size cố định	Kích thước có thể tăng
Rust Primitives	Collections (HashMap, Vec , String, ..)

2. Ownership

```
// let s1 = String::from("hello");
// let s2 = s1;
```

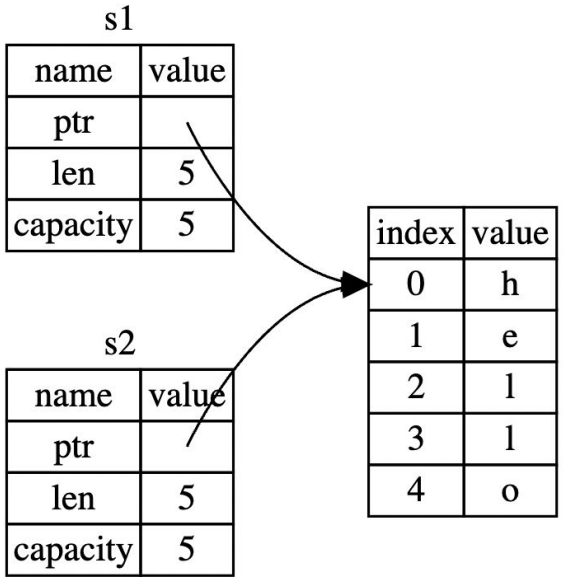


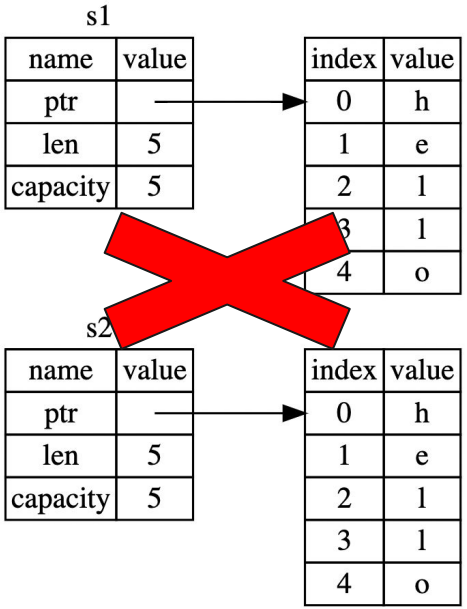
+ Lưu pointer, length , capacity trên Stack
+ Dữ liệu “hello" lưu trên Heap

Phân biệt giữa len và capacity?

2. Ownership

Khi gán s1 cho s2



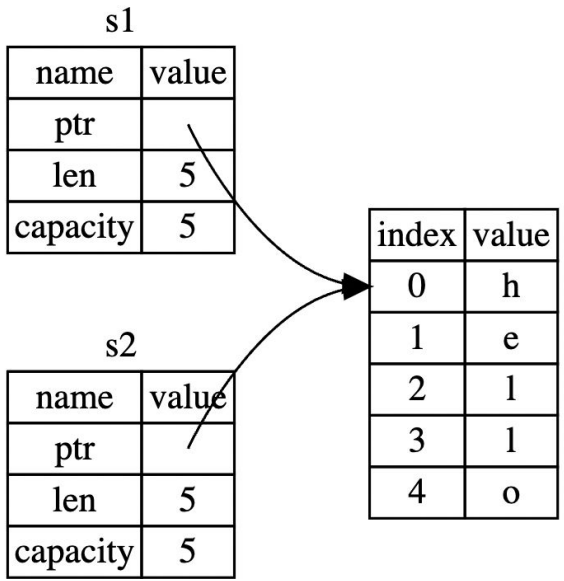


2. Ownership

Có vấn đề xảy ra như sau:

+ Khi s1 và s2 out of scope -> double free error -> ko an toàn

```
    let s1 = String::from("hello");
        -- move occurs because `s1` has type `String`, which does not implement
`Copy` trait
    let s2 = s1;
             -- value moved here
    println!("s1: {}", s1);
                       ^^ value borrowed here after move
```



Ownership

2. Ownership

Các quy tắc của Ownership:

1. Mỗi biến lưu ứng với 1 giá trị sẽ có đặc tính owner
2. Chỉ có 1 owner tại 1 thời điểm
3. Khi biến owner thoát ra khỏi scope, thì giá trị sẽ bị drop theo

3. Reference

Làm sao để giải quyết trường hợp muốn sử dụng lại biến owner ?

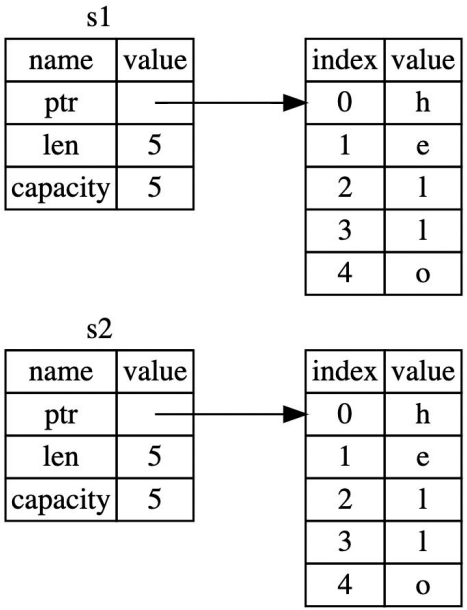
```
    let s1 = String::from("hello");
        -- move occurs because `s1` has type `String`, which does not implement
`Copy` trait
    let s2 = s1;
             -- value moved here
    println!("s1: {}", s1);
                       ^^ value borrowed here after move
```

Cách 1: Clone

Cách 2: Reference

3. Reference

Clone




```
let s1: String = String::from("hello");
let s2: String = s1.clone();
println!("s1: {}", s1);
println!("s2: {}", s2);
```

Dùng function/method clone()

Nhược điểm?

3. Reference

```
fn main() {
    let s = String::from("hello");
   let s1 = &s;
}
```

3. Reference

Shared Reference

+ Giá trị khi mà shared reference sẽ không thể nào thay đổi (immutable)
+ Không thể sửa đổi hoặc chỉ định lại giá trị mà một tham chiếu được chia sẻ trỏ tới
+ Không thể cast từ shared reference thành mutable

3. Reference

```
fn main() {
    let s = String::from("hello");
    change(&s);
}
fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
|     some_string.push_str(", world");
|     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the d
it refers to cannot be borrowed as mutable
```

3. Reference

Mutable Reference

+ Biến tham chiếu có thể thay đổi được

```
fn test_mutable(input:&u32, output: &mut u32) -> u32{
if *input ==1 { *output =3;    }
else { *output = 4;   }
*output }
```

3. Reference

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

3. Reference

Dangling reference

+ Xảy ra khi con trỏ tham chiếu tới vùng nhớ mà giá trị lưu trữ ở vùng nhớ này bị xoá đi trong khi con trỏ vẫn tham chiếu tới vùng nhớ mà đã bị xoá (null pointer)
+ Compiler ở Rust sẽ kiểm tra trong trường hợp sau: nếu bạn có tham chiếu tới 1 giá trị nào đó , thì compiler sẽ đảm bảo rằng giá trị sẽ ko bị drop trước khi bạn sử dụng biến tham chiếu

3. Reference

Một số giới hạn của reference

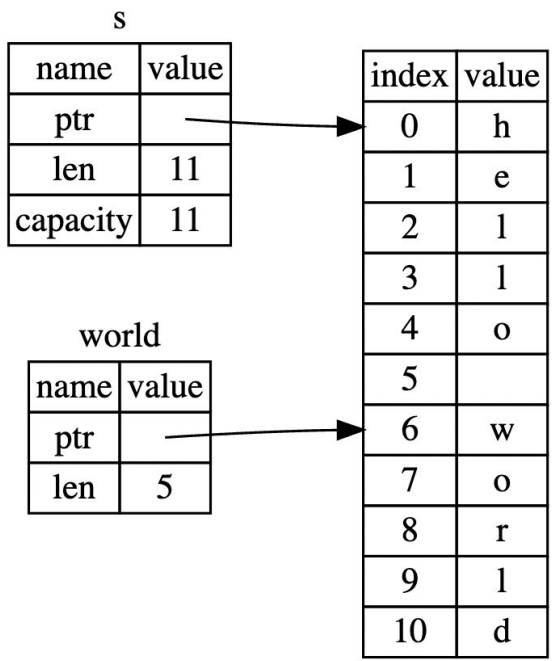
+ chỉ 1 mutable reference cho 1 giá trị nhất định trong 1 thời điểm
+ Nhiều immutable reference được cho phép sử dụng (đọc)
+ Không thể có mutable reference trong khi có 1 immutable reference tham chiếu tới cùng 1 giá trị

3. Reference

Phân biệt Slice string và String

```
fn main() {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```



3. Reference

Slice String	String
Kích thước cố định	Không xác định được kích thước
Immutable	Mutable

4. Thực hành

https://github.com/CocDap/VBI-Rust-Substrate-3