

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





Workshop 2: Ownership and Borrowing

VBI Academy



1. 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



1. Ownership

Để hiểu giải thích vấn đề trên cần tìm hiểu 3 vấn đề nữa

- + Phân biệt biến , pointer
- + Compile time vs runtime
- + Stack/Heap



1. Ownership

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

1. Ownership

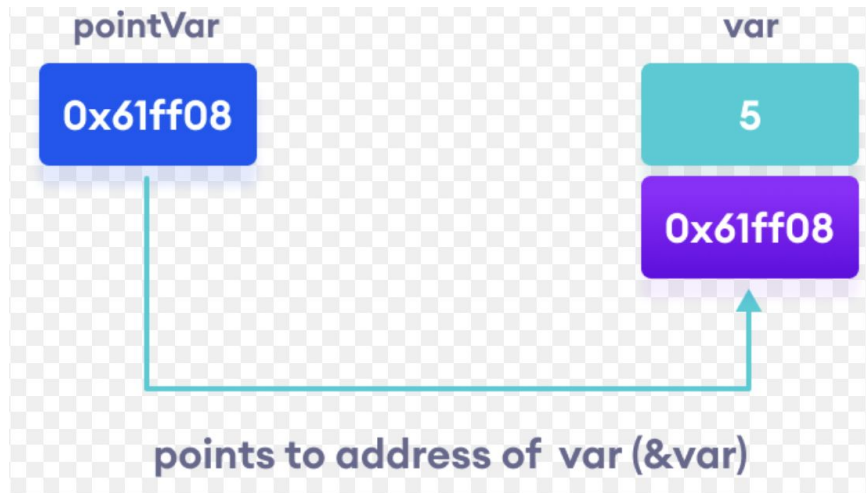
let x = 42;

let y = 43;

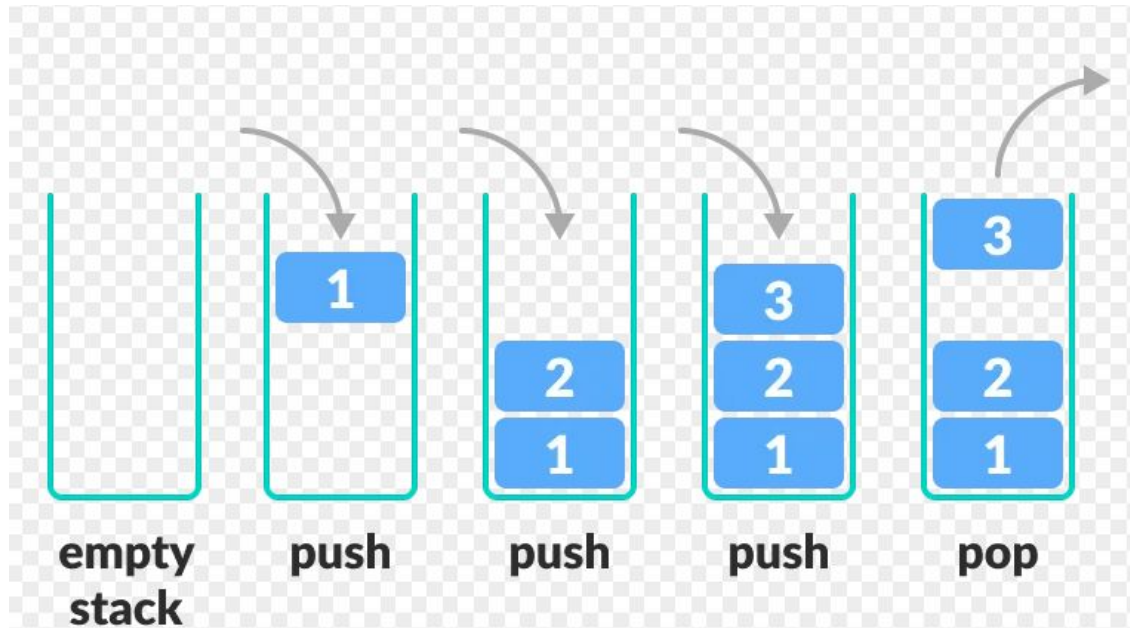
let var1 = &x;

let mut var2 = &x;

var2 = &y



1. Ownership



1. Ownership

Stack

```
fn foo() {  
    let y: i32 = 5;  
    let z: i32 = 100;  
}  
  
fn main() {  
    let x: i32 = 42;  
    foo();  
}
```

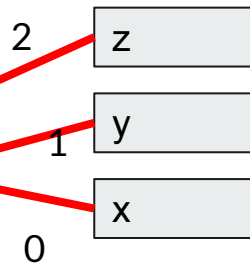
Address	Name	Value
2	z	100
1	y	5
0	x	42

Khi hết scope
của hàm foo()

Address	Name	Value
0	x	42

Khi hết scope
của hàm main()

Stack rỗng





1. Ownership

Heap

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

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



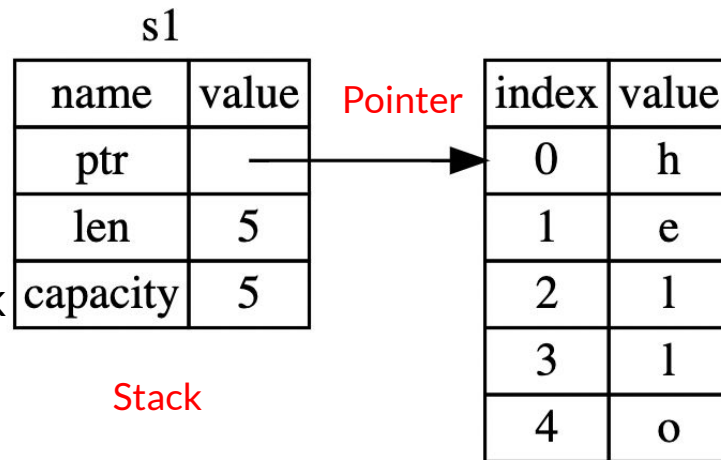
1. Ownership

Stack	Heap
Việc tạo vùng nhớ cho giá trị, việc đọc giá trị nhanh	Runtime performance 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, ..)

1. Ownership

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

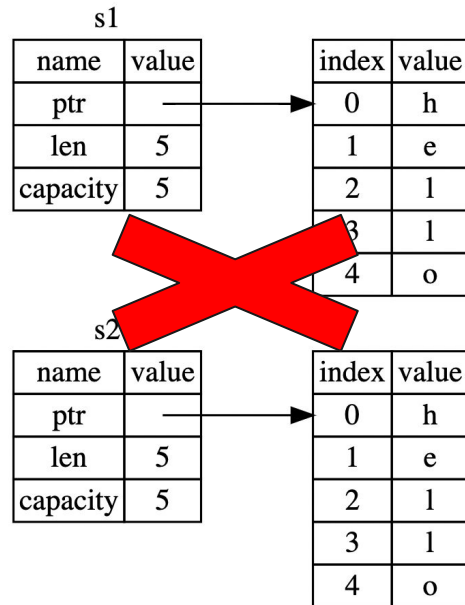
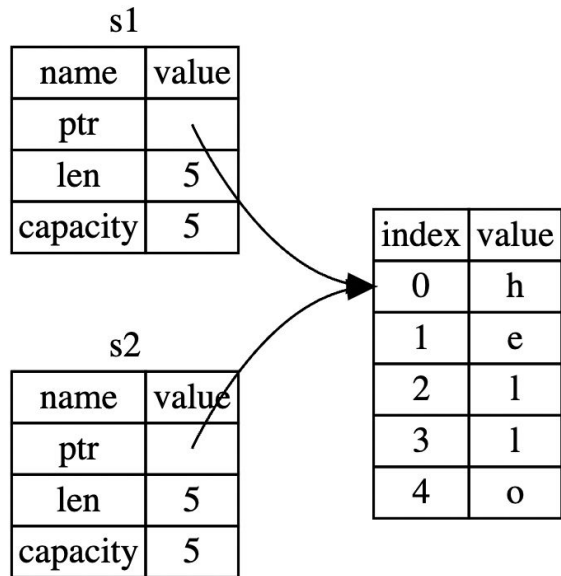
- + Lưu pointer, length , capacity trên Stack
- + Dữ liệu "hello" lưu trên Heap



Phân biệt giữa len và capacity?

1. Ownership

Khi gán s1 cho s2



1. Ownership

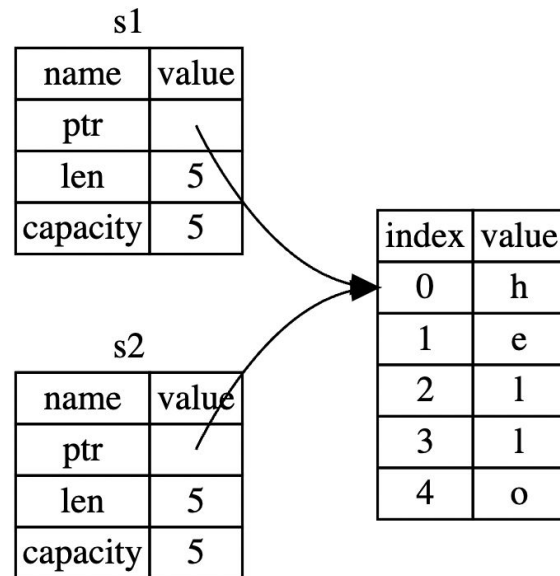
Có vấn đề xảy ra như sau:

- + 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);  
    ^^ value borrowed here after move
```

Ownership





1. 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



2. Reference

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

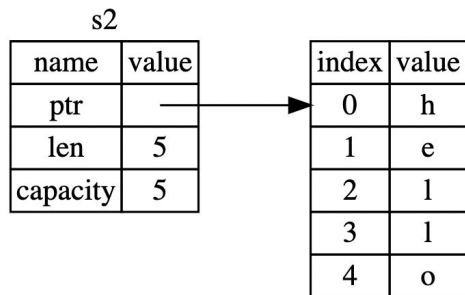
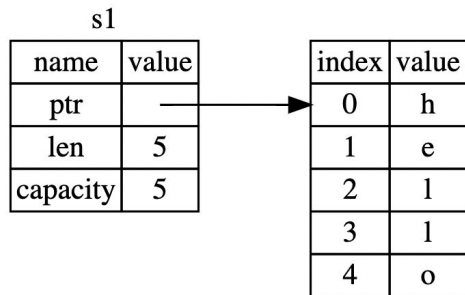
```
let s1 = String::from("hello");  
    -- move occurs because `s1` has type `String`, which does not implement  
`Copy` trait  
let s2 = s1;  
    -- value moved here  
println!("s1: {}", s1);  
    ^^ value borrowed here after move
```

Cách 1: Clone

Cách 2: Reference

2. 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?



2. Reference/ Borrowing

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



2. 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 trở tới
- + Không thể cast từ shared reference thành mutable



2. Reference

```
fn main() {
```

```
    let s = String::from("hello");
```

```
    change(&s);
```

```
}
```

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

```
fn change(some_string: &String) {
```

```
    some_string.push_str(", world");
```

```
}
```



2. 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 }
```



2. Reference

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

<https://users.rust-lang.org/t/quick-question-static-str/35940/3>



2. Reference

Dangling reference

- + Xảy ra khi con trỏ tham chiếu tới vùng nhớ mà giá trị lưu trữ ở vùng nhớ này bị xóa đi trong khi con trỏ vẫn tham chiếu tới vùng nhớ mà đã bị xóa (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



2. 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

2. Reference

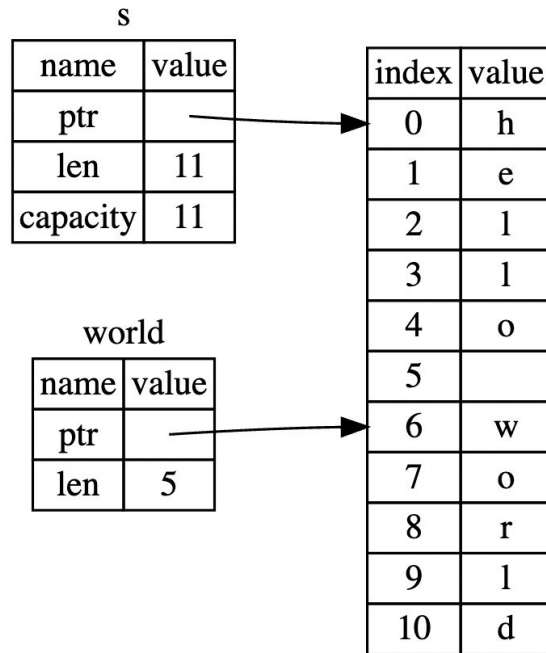
Phân biệt Slice string và String

```
fn main() {
```

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

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

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





2. Reference

Slice String	String
Kích thước cố định ở compile time	Không xác định được kích thước ở compile time
Immutable	Mutable
Stack	Heap