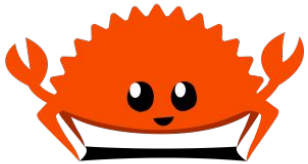
Manejo de memoria

NACION CRYPTO By DECRY

Memoria

Rust es un lenguaje de programación con manejo seguro de memoria. Para lo cual introduce conceptos como:

- Stack.
- Heap..
- OwnerShip (propiedad).
- Referencia (reference)
- Prestamo (borrowing)

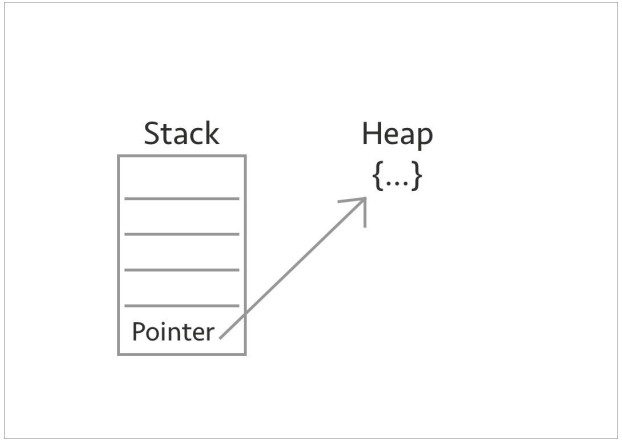


Stack

Caracteristicas:

- Almacena la información de forma ordenada.
- Hay mayor velocidad
- Toda la data debe ser conocida en su tamaño.
- Inserta la información en orden como la recibe. (Pushing on the stack)
- Datos primitivos y variables locales de las funciones son alojadas aquí.
- Remueve la información de manera opuesta. (popping off the stack)

LIFO

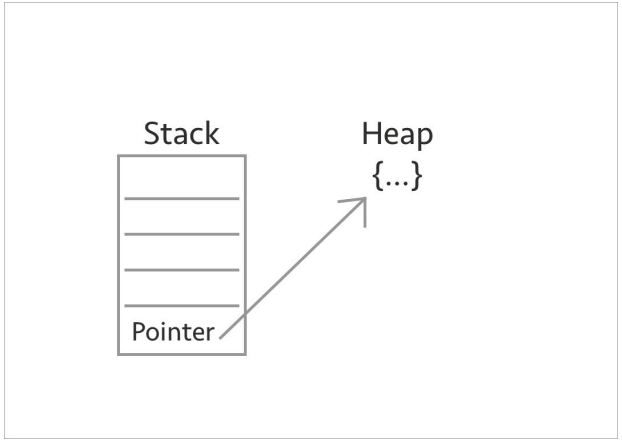


Stack

Caracteristicas:

- Almacena la información de forma ordenada.
- Toda la data debe ser conocida en su tamañoInserta la información en orden como la recibe. (Pushing on the stack)
- Remueve la información de manera opuesta. (popping off the stack)

LIFO



Stack

```
fn foo() {
   let y = 999;
   let z = 333;
}

fn main() {
   let x = 111;

   foo();
}
```

1. Cuando se llama la función Main

Address	Name	Value
0	x	111

2. Cuando se llama la función foo

Address	Name	Value
2	z	333
1	y	999
0	x	111

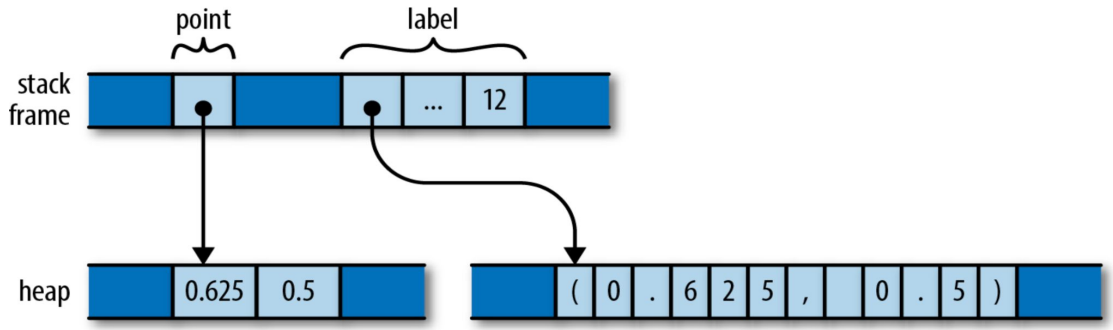
3. Después que se ejecute la función foo

Address	Name	Value
0	x	111

Heap

Caracteristicas:

- Es menos organizado.
- Cuando se solicita espacio, el allocator asigna un fragmento a la información que sea lo suficientemente grande. La marca como que está en uso y retorna un puntero.
- Tipos de datos dinámicos son alojados en el heap, como son String, Vector, Box, etc.



Heap

```
fn main() {
    let x = Box::new(100);
    let y = 222;

    println!("x = {}, y = {}", x, y);
}
```

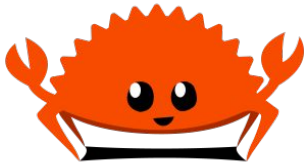
Address	Name	Value
5678		100
...	...	...
1	y	222
0	x	→ 5678

Ambito ó Scope

```
fn nombre ( argumentos: tipo)  {
        let var = valor;
}
```

```
fn main() {
   // El Ambito de esta variable es la funcion main
   Let edad = 36;
   …
}
```

El Ámbito de una variable es la región/bloque del código donde ella está disponible. Una función también es un ámbito.

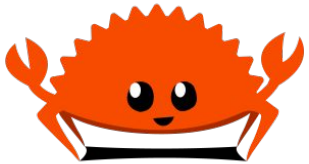


Ambito ó Scope

```
fn main() {
 let x: i32 = 17;

{

let y: i32 = 3;
println!("The value of x is {} and value of y is {}", x, y);
}
println!("The value of x is {} and value of y is {}", x, y); // Esto no compila.
}
```

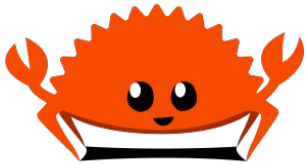
```
fn main() {
        let x: i32 = 8; {
        println!("{}", x); // Imprime "8".
        let x = 12;
        println!("{}", x); // Imprime "12".
        }
        println!("{}", x); // Imprime "8".
        let x = 42;
        println!("{}", x); // Imprime "42".

}
```

Si tratamos de compilar hay error, porque la variable Y solo es válida dentro de su ámbito.



Shadowing

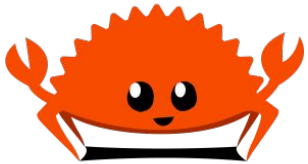
```
fn main() {
    let random = 100;
    // start of the inner block
    {
        println!("random variable before shadowing in inner block = {}", random);
        // this declaration shadows the outer random variable
        let random = "abc";
        println!("random after shadowing in inner block = {}", random);
    }
    // end of the inner block
    println!("random variable in outer block = {}", random);
}
```



Propietario (OwnerShip)

Reglas:

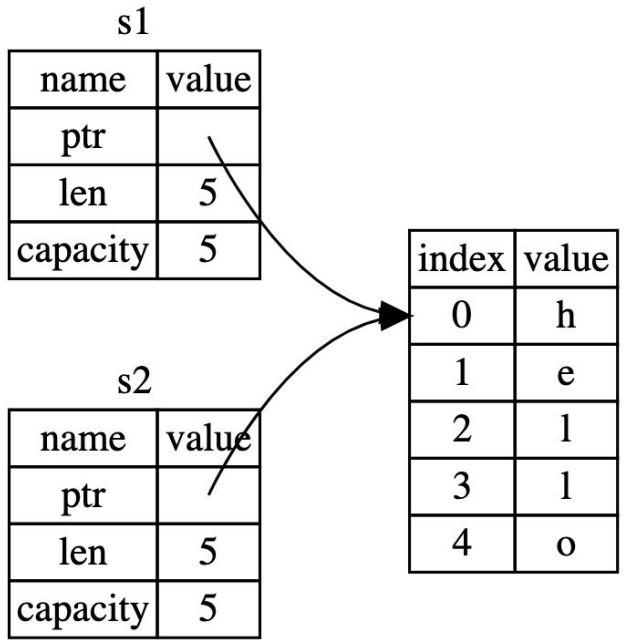
- Cada valor en Rust tiene una variable, su propietario.
- Solo hay un propietario a la vez.
- Cuando el propietario sale del ámbito, la variable se borra.



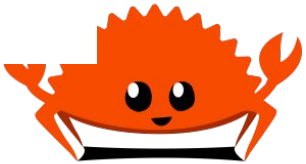
OwnerShip

MOVE

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



No se puede imprimir el valor de s1 después de moverlo.

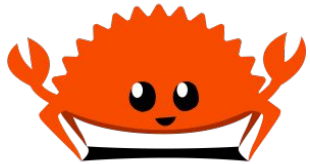


OwnerShip

MOVE

```
    fn main() {
    let x = 11;
    // copia los datos de x a y
    // La regla del ownership no aplica aquí
    let y = x;

    println!("x = {}, y = {}", x, y);
}
```

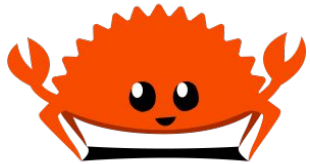
Esta copia es posible porque son datos primitivos (tamaño conocido por el compilador)



MOVE EN FUNCIONES

```
fn process(input: String) {}

fn caller() {
    let s = String::from("Hello, world!");
    process(s); // Ownership of the string in `s` moved into `process`
    process(s); // Error! ownership already moved.
}
```

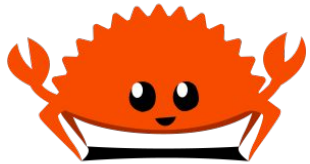
La propiedad se transfiere a la función



COPY EN FUNCIONES

```
fn process(input: u32) {}

fn caller() {
   let n = 1u32;
   process(n); // Ownership of the number in `n` copied into `process`
   process(n); // `n` can be used again because it wasn't moved, it was copied.
}
```

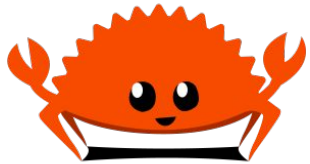
La propiedad se copia dado que es un entero y sigue estando en el ámbito.



COPIAR DATOS DE VARIABLES COMPLEJAS

```
fn process(s: String) {}

fn main() {
    let s = String::from("Hello, world!");
    process(s.clone()); // Passing another value, cloned from `s`.
    process(s); // s was never moved and so it can still be used.
}
```

Duplica la memoria y genera un nuevo valor, mediante el uso del método clone



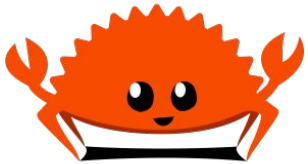
Prestamos

Estaría bien que una función y variables usaran los datos sin tener que ser dueños?

Después de que una variable sea referenciada por otras variables, la propiedad de su valor permanece y no se pierde. ***Por defecto una referencia siempre es inmutable.***

Referencia a una Variable: **&**variable

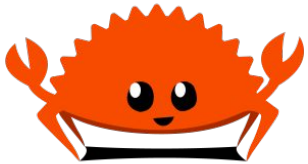
Referencia a un Argumento o Parámetro. (Préstamo)
parameter : **&**type



Prestamos

Ejemplo

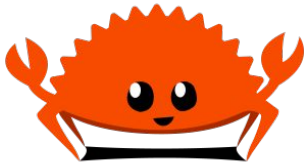
```
fn print_greeting(message: &String) {
  println!("Greeting: {}", message);
}

fn main() {
  let greeting = String::from("Hello");
  print_greeting(&greeting); // `print_greeting` takes a `&String` not an owned `String` so we borrow `greeting` with `&`
  print_greeting(&greeting); // Since `greeting` didn't move into `print_greeting` we can use it again
}
```



Prestamos

Ejemplo

```
fn main() {
    let mut str = String::from("Hello");
    // before modifying the string
    println!("Before: str = {}", str);
    // pass a mutable string when calling the function
    change(&mut str);
    // after modifying the string
    println!("After: str = {}", str);
}

fn change(s: &mut String) {
    // push a string to the mutable reference variable
    s.push_str(", World!");
}
```



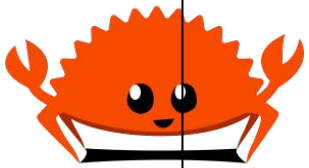
Prestamos

Regla:

El código debe implementar *cualquiera* de las definiciones siguientes, pero no las dos al mismo tiempo:

- Una o más referencias inmutables (&T)
- Exactamente una referencia mutable (&mut T)

```
fn main() {
    let mut str = String::from("hello");

    // mutable reference 1
    let ref1 = &mut str;

    // mutable reference 2
    let ref2 = &mut str;

    println!("{}, {}", ref1, ref2);
}
```



BIBLIOGRAFIA:

- https://www.oreilly.com/library/view/programming-rust/9781491927274/ch04.html
- https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap
- https://www.youtube.com/watch?app=desktop&v=rDoqT-a6UFg