Ownership y Borrowing

Ownership y borrowing

Para el manejo de memoria de los programas hay 2 enfoques que utilizan mucho de los lenguajes más usados:

-Tener un garbage collector que busca periódicamente memoria que no se use para limpiarla.

-Y otro enfoque donde se debe asignar y liberar memoria explícitamente.

Rust usa un tercer enfoque, la memoria se administra a través de un concepto de propiedad.

El concepto de ownership refiere a un conjunto de reglas de como Rust maneja la memoria

Ownership y borrowing

Estas reglas son las siguientes:

1. Cada valor en Rust tiene un dueño.
2. Solo puede haber un dueño a la vez.
3. Cuando el dueño queda fuera del alcance, el valor se eliminará.

Ownership y borrowing

Stack vs Heap:

La memoria stack es rápida, es liberada cuando se alcanza el fin del scope: aqui irán los datos de tipo de tamaño conocido en tiempo de compilación como por ej i32.

La memoria heap es flexible, tiene elevado costo en asignar y recuperar datos. Es liberada cuando no tiene dueños. Aquí irán los datos de tipo de tamaño desconocido en tiempo de compilación como ser String.

Ownership y borrowing

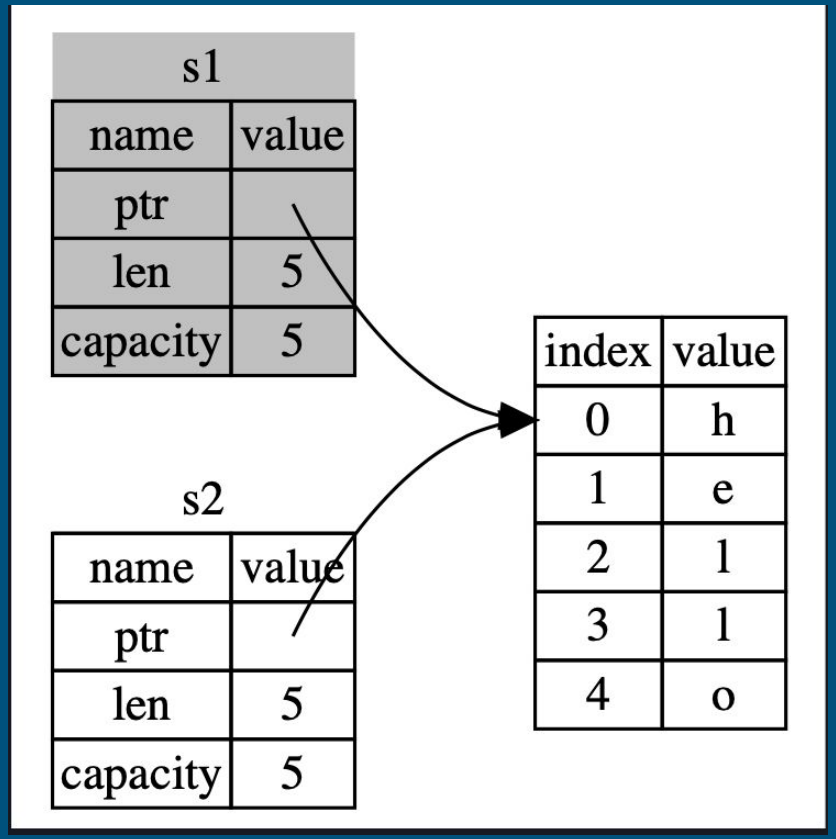
```
fn main() {
    let s1= 10;
    let s2 = s1;
    println!("{}", s1);
}
```

Ownership y borrowing

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

```
error[E0382]: borrow of moved value: `s1`
  --> src/main.rs:14:20
   |
12 |     let s1= String::from("data ");
   |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
13 |     let s2 = s1;
   |              -- value moved here
14 |     println!("{}", s1);
   |                    ^^ value borrowed here after move
   |
```

Ownership y borrowing



Ownership y borrowing

Que tipos implementan el trait Copy:

- Todos los enteros
- Booleanos
- Punto flotante
- Char
- Tupla que solo tengan los tipos que implementan Copy

Ownership y borrowing: funciones

```
fn main() {
    let mut dato1= 10;
    mi_funcion(dato1);
    println!("{}", dato1);
}

fn mi_funcion(mut data: i32){
    data+=1;
    println!("muestro data en la funcion:  {}", data);
}
```

Ownership y borrowing: funciones

```
fn main() {
    let mut dato1= 10;
    mi_funcion(&mut dato1);
    println!("{}", dato1);
}

fn mi_funcion(data: &mut i32){
    *data+=1;
    println!("muestro data en la funcion: {}", data);
}
```

Ownership y borrowing: funciones

```
fn main() {
    let dato1= String::from(" Seminario de: ");
    mi_funcion(dato1);
    println!("{}", dato1);
}
fn mi_funcion(data: String){
    println!("muestro data en la funcion: {}", data);
}
```

Ownership y borrowing: funciones

```
error[E0382]: borrow of moved value: `dato1`
  --> src/main.rs:14:20
   |
12 |     let dato1= String::from(" Semnario de: ");
   |         ----- move occurs because `dato1` has type `String`, which does not implement the `Copy` trait
13 |     mi_funcion(dato1);
   |                ----- value moved here
14 |     println!("{}", dato1);
   |                    ^^^^^ value borrowed here after move
   |
note: consider changing this parameter type in function `mi_funcion` to borrow instead if owning the value i
sn't necessary
  --> src/main.rs:17:22
   |
17 |  fn mi_funcion(data: String){
   |     ----------       ^^^^^^ this parameter takes ownership of the value
   |     |
   |     in this function
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the
 macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
   |
13 |     mi_funcion(dato1.clone());
   |                     ++++++++
```

Ownership y borrowing: funciones

```
fn main() {
    let dato1= String::from(" Seminario de: ");
    mi_funcion(&dato1);
    println!("{}", dato1);
}
fn mi_funcion(data: &String){
    println!("muestro data en la funcion: {}", data);
}
```

Ownership y borrowing: funciones

```
fn main() {
    let dato1= String::from(" Seminario de: ");
    let dato1 = mi_funcion(dato1);
    println!("{}", dato1);
}
fn mi_funcion(data: String) -> String{
    println!("muestro data en la funcion: {}", data);
    data
}
```

Ownership y borrowing: funciones

```
fn main() {
    let mut dato1= String::from(" Seminario de: ");
    mi_funcion(&mut dato1);
    println!("{}", dato1);
}
fn mi_funcion(data: &mut String){
    data.push_str(" Rust!");
    println!("muestro data en la funcion: {}", data);
}
```

Lifetime

Lifetime

Cada referencia en Rust tiene una vida útil, que es el alcance para el cual esa referencia es válida.

La mayoría de las veces el tiempo de vida de las referencias se infieren al igual que la mayoría de las veces se infieren los tipos.

Lifetime es la manera que tiene el compilador de Rust de asegurar que un lugar de memoria es válido para una referencia.

Lifetime: ejemplos

```rust
fn main() {
    let dato1: &i32;
    {
        let otro_scope = 2;
        dato1 = &otro_scope;
    }
    println!("{}", dato1);
}
```

```
error[E0597]: `otro_scope` does not live long enough
  --> src/main.rs:15:17
   |
15 |         dato1 = &otro_scope;
   |                 ^^^^^^^^^^^ borrowed value does not live long enough
16 |
17 |     }
   |     - `otro_scope` dropped here while still borrowed
18 |     println!("{}", dato1);
   |                    ----- borrow later used here
```

Lifetime: ejemplos

```
fn main() {
    let d1 = "str1";
    let d2 = "str2";
    let r = crear(d1, d2);
    println!("{}", r.as_str());
}
fn crear(data1: &str, data2: &str)-> &String{
    let resultado:String = data1.to_string().add(data2);
    &resultado
}
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:18:38
   |
18 | fn crear(data1: &str, data2: &str)-> &String{
   |                 ----          ----     ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature does not say whether it
is borrowed from `data1` or `data2`
help: consider introducing a named lifetime parameter
   |
18 | fn crear<'a>(data1: &'a str, data2: &'a str)-> &'a String{
   |         ++++         ++              ++         ++
```

Lifetime: ejemplos

```rust
use std::{ops::Add};
fn main() {
    let d1 = "str1";
    let d2 = "str2";
    let r = crear(d1, d2);
    println!("{}", r);
}
fn crear<'a>(data1: &'a str, data2: &'a str)-> &'a str{
    let d1 = data1.to_string();
    let resultado:&str = d1.add(data2).as_str();
    &resultado
}
```

```
error[E0515]: cannot return value referencing temporary value
  --> src/main.rs:11:5
   |
10 |     let resultado:&str = d1.add(data2).as_str();
   |                          ------------- temporary value created here
11 |     &resultado
   |     ^^^^^^^^^^ returns a value referencing data owned by the current function
```

Lifetime: ejemplos

```
fn main() {
    let string1 = String::from("Seminario de:");
    let string2 = "Rust!!!";
    let result = mas_largo(string1.as_str(), string2);
    println!("El mas largo es:{}", result);
}
fn mas_largo(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:35
  |
9 | fn mas_largo(x: &str, y: &str) -> &str {
  |                 ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `
x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {
  |             ++++     ++          ++          ++
```

Lifetime: ejemplos

```
fn main() {
    let string1 = String::from("Seminario de:");
    let string2 = "Rust!!!";
    let result = mas_largo(string1.as_str(), string2);
    println!("El mas largo es:{}", result);
}
fn mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime: sintaxis

```
&i32        // una referencia
&'a i32     // una referencia con explicito lifetime
&'a mut i32 // una referencia mutable con explicito lifetime
```