Rust, el lenguaje

ACM, Pepe Márquez Romero

Rust, el lenguaje

@acmupm

Guía de instalación de Rust

13 de febrero de 2022

Índice

Tipos básicos de Rust I

Integer types		
Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Tipos básicos de Rust II

Integer literals types	
Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Bytes (u8 only)	b'A'

Tipos básicos de Rust III

Float number types	
32-bits	f32
64-bits	f64

Primitive string and char	
str	char

Algunos comandos de cargo

```
$ cargo new acm_rust

$ cargo build

$ cargo run

$ cargo test

$ cargo doc --open
// Mencion honorifica a rustup

$ rustup toolchain install nightly

$ rustup override set nightly
```

Declaración de variables

```
let x : u8 = 32 ; // variable especificando tipo
let y = 32 ; // infiere el tipo i32
let mut z : f32 = 3.4;// variable mutable
let mut n = 43.77;//variable mutable infiere tipo f64
const PI : f64 = 3.14159;
// se evalua en tiempo de compilacion
```

Ejemplo básico

Un poco de código para familiarizarse con Rust

Vamos a ver con estos ejemplos algunas cosas de Rust que hace que sea un poco diferente a otros lenguajes.

```
fn main (){
  let x :u8 = 22;
  println!("Hello World :D !");
  let s : &str = "hello"; // ojo dos tipos de string
  let st : String = String::from("world");
  let mut x : u32 = 74; // ojo al shadowing
  println!("{}" , x);// imprime 74
  x = 42;// mutable gracias shadowing
  println!("{}" , x);// imprime 42
}
```

¿Qué es el Ownership?

Ownership

Sistema de Rust para saber cuando debe liberar la memoria de una instancia que se realiza con la llamada del método drop.

¿En qué consiste el Ownership?

Se basa, en que toda instancia tiene dueño, y que el dueño es el ámbito donde se ha declarado la instancia y cuando termina ese ámbito se liberan las instancias, que no han sido movidas a otros ámbitos, es decir, que no han cambiado de dueño.

Reglas del Ownership

1. Each value in Rust has a variable that's called its owner.

Reglas del Ownership

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.

Reglas del Ownership

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Consecuencias del Ownership

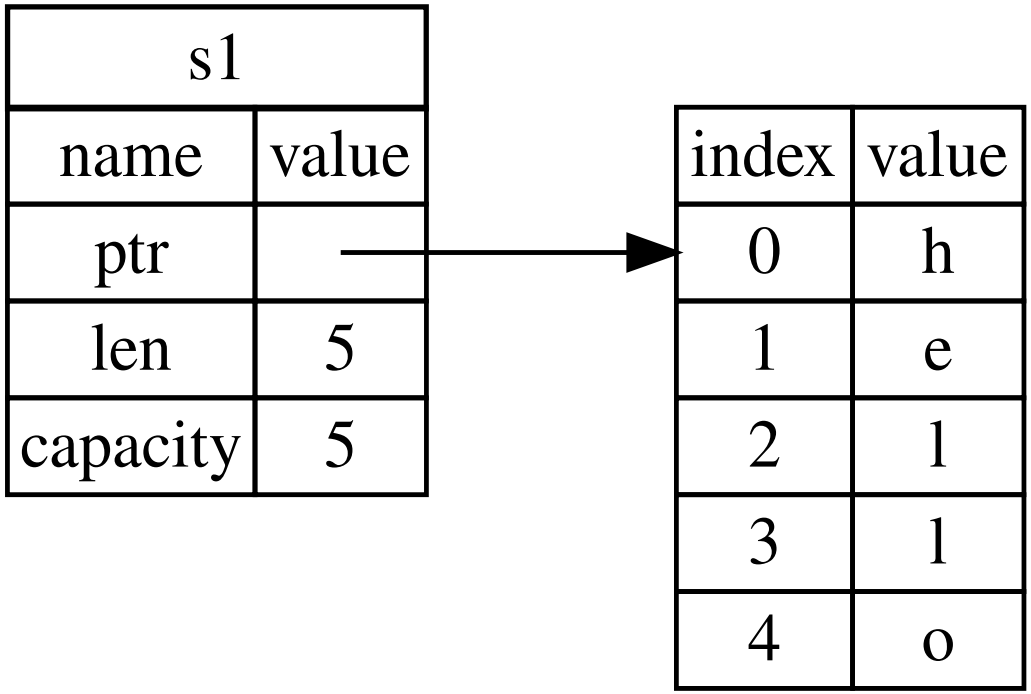


Figura: String en Rust.

Consecuencias del Ownership

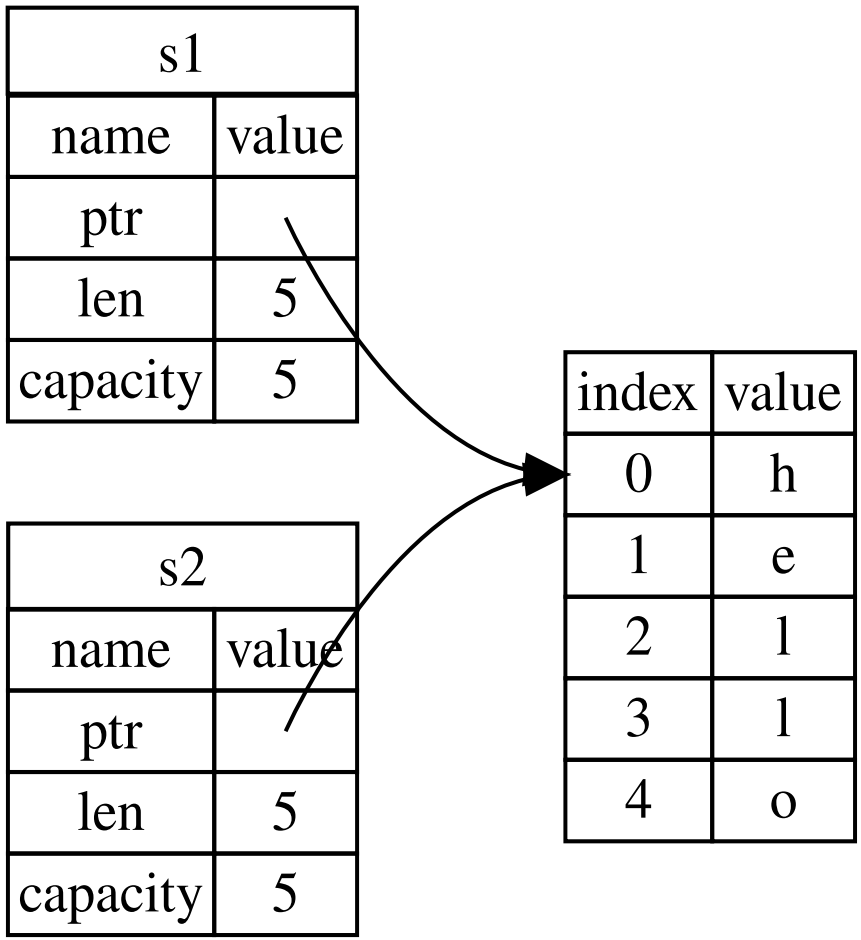


Figura: Soft copy.

Consecuencias del Ownership

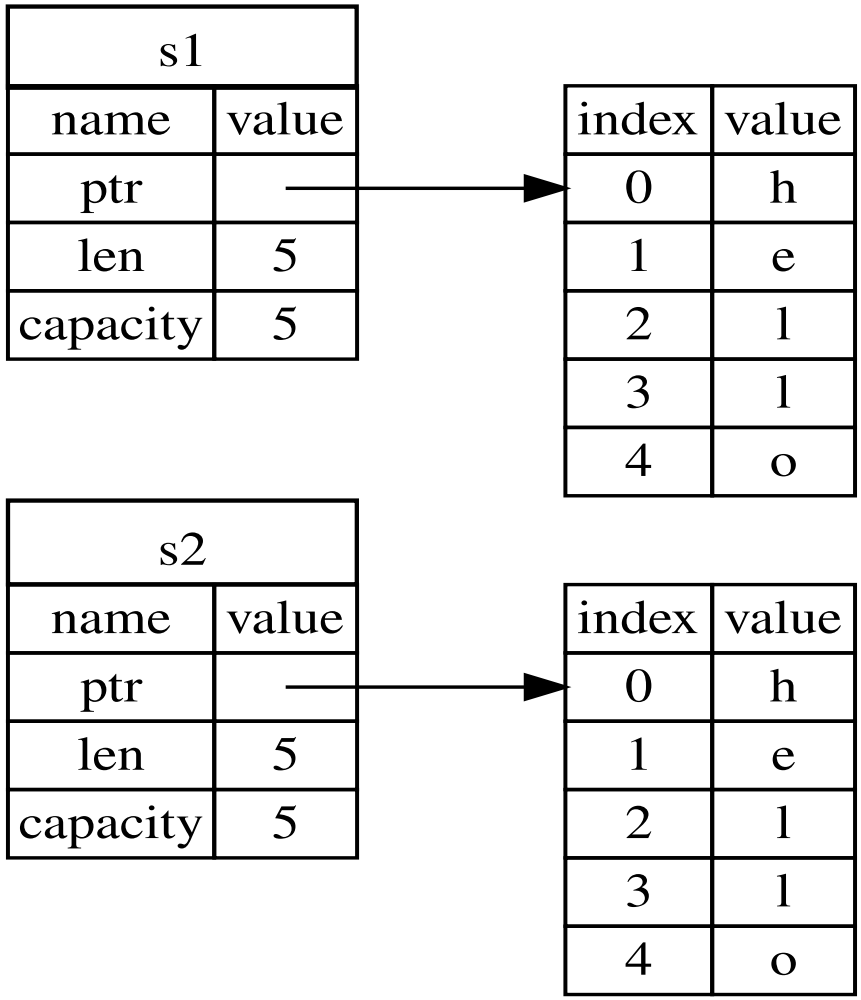


Figura: Deep copy.

Consecuencias del Ownership

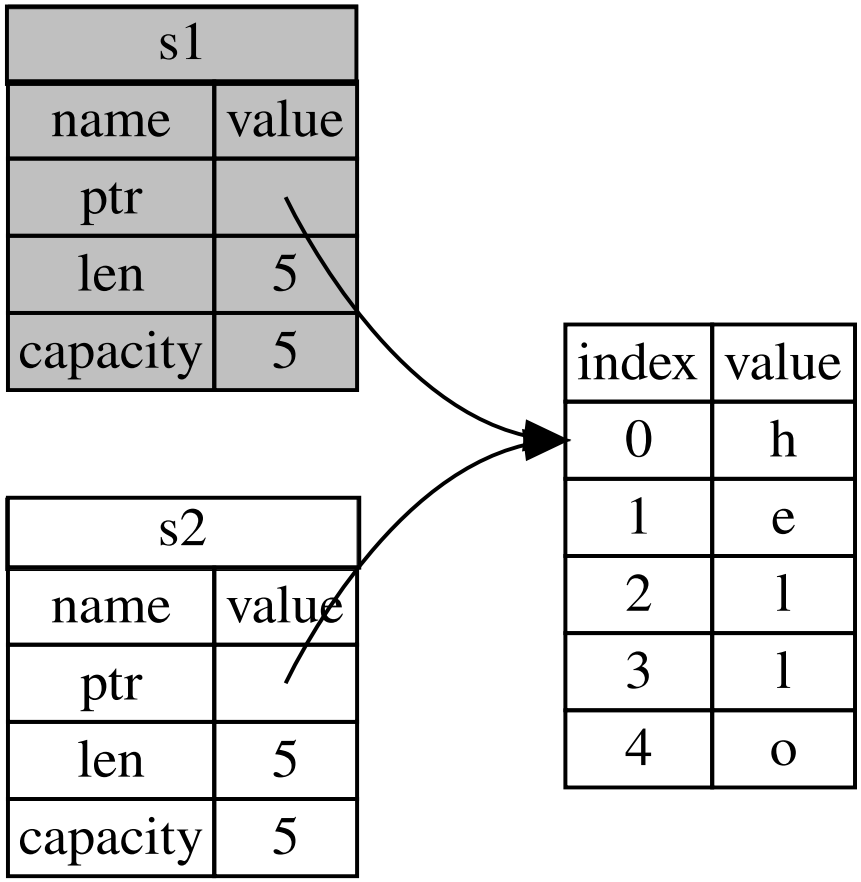


Figura: Lo que hace Rust cuando no tiene el Copy Trait implentado.

Ejemplo donde el Ownership afecta al código

```
1  fn its_mine_haha(s : String){
2    println!("\"{}\" its mine !" , s);
3  }
4  fn main (){
5    let x :u8 = 22;
6    let y = x ; // copia el valor de x en y
7    // Ojo al Copy Trait
8    println!("{}" , x);// imprime 22
9    println!("{}" , y);// imprime 22
10   let z : String = String::from("hi");
11   let w = z ;
12   println!("{}" , w);// Ojo z se ha movido a w
13   //luego z no se puede usar
14   its_mine_haha(w);// w se mueve a la funcion
15   // luego w no se puede usar aqui
16 }
```

Borrowing

¿Qué es el Borrowing?

Es el mecanismo que permite que puedas usar ciertas instancias sin capturar el owner de esa instancia, para que siga viviendo en el ámbito de donde viene.

Reglas del Borrowing

1. At any given time, you can have either one mutable reference or any number of immutable references.

Reglas del Borrowing

1. At any given time, you can have either one mutable reference or any number of immutable references.
2. References must always be valid.

Consecuencias del Borrowing

1. Eliminación de los data races.

Consecuencias del Borrowing

1. Eliminación de los data races.
   1.1 Two or more pointers access the same data at the same time.

Consecuencias del Borrowing

1. Eliminación de los data races.
   - 1.1 Two or more pointers access the same data at the same time.
   - 1.2 At least one of the pointers is being used to write to the data.

Consecuencias del Borrowing

1. Eliminación de los data races.
   - 1.1 Two or more pointers access the same data at the same time.
   - 1.2 At least one of the pointers is being used to write to the data.
   - 1.3 There's no mechanism being used to synchronize access to the data.

Consecuencias del Borrowing

1. Eliminación de los data races.
   - 1.1 Two or more pointers access the same data at the same time.
   - 1.2 At least one of the pointers is being used to write to the data.
   - 1.3 There's no mechanism being used to synchronize access to the data.

2. Eliminación de las Dangling References.

Ejemplos de Borrowing

```
1 fn out_print(s: &String) {
2   println!("\"{}\" its printed from out_print !", s);
3 }
4 fn main() {
5   let mut z: String = String::from("hi");
6   out_print(&z);
7   z = String::from("hello");
8   //*(&mut z) = String::from("hello");
9   //let h: &mut String = &mut z;
10  //let j: &mut String = &mut z;
11  println!("\"{}\" its printed from main!", z);
12  z.push_str("_there");
13  println!("\"{}\" its printed from main!", z);
14 }
```

Rich Pattern Matching.

Formas de hacer pattern matching

1. Sentencia match.

Rich Pattern Matching.

Formas de hacer pattern matching

1. Sentencia match.
2. Setencia if let.

Rich Pattern Matching.

Formas de hacer pattern matching

1. Sentencia match.
2. Setencia if let.
3. Sentencia while let .

Rich Pattern Matching.

Formas de hacer pattern matching

1. Sentencia match.
2. Setencia if let.
3. Sentencia while let .
4. Macro matches! .

Enumerales que enriquecen los patrones

```rust
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
pub enum Option<T> {
    None,
    Some(T),
}
```

Ejemplos de Pattern Matching I

```
1 use std::env;
2 use std::process;
3 fn main() {
4   let mut args = env::args().skip(1);
5   //std::iter::Skip<env::Args>
6   let arg : Option<String> = args.next();
7   let some_st: String;
8   if let None = arg {
9     eprintln!("Me tienes que llamar con un
10    entero sin signo de 8 bits");
11    process::exit(3);
12  } else {
13    some_st = arg.unwrap(); //inicializando el valor
14  }
15  println!("Tratamos de parsear {}", some_st);
```

Ejemplos de Pattern Matching II

```
  match some_st.parse::<u8>() {
    Ok(1) => println!("Es uno :D"),
    Ok(2) => println!("Es dos :D"),
    Ok(value) => println!("Es {} :D", value),
    Err(e) => {
      eprintln!("No se puede pasear por {}", e);
      process::exit(2);
    }
  }
}//llave del main
```

Ejemplos de Pattern Matching III

```
use std::io::stdin
fn main() {
  let vector:Vec<String> = vec!["hi".to_string(),
    "hello".to_string(), "there".to_string()];
  let mut iter1 = vector.iter().peekable();
  while let Some(_) = iter1.peek() {
    println!("{}", iter1.next().unwrap());
  }
  let mut string: String = String::new();
  while matches!(stdin().read_line(&mut string), Ok(_))
    && !string.eq("") {
    // haz algo con la linea de stdin
    string = String::from("");
  }
}
```

Funciones anónimas (Closures) I

1. FnOnce

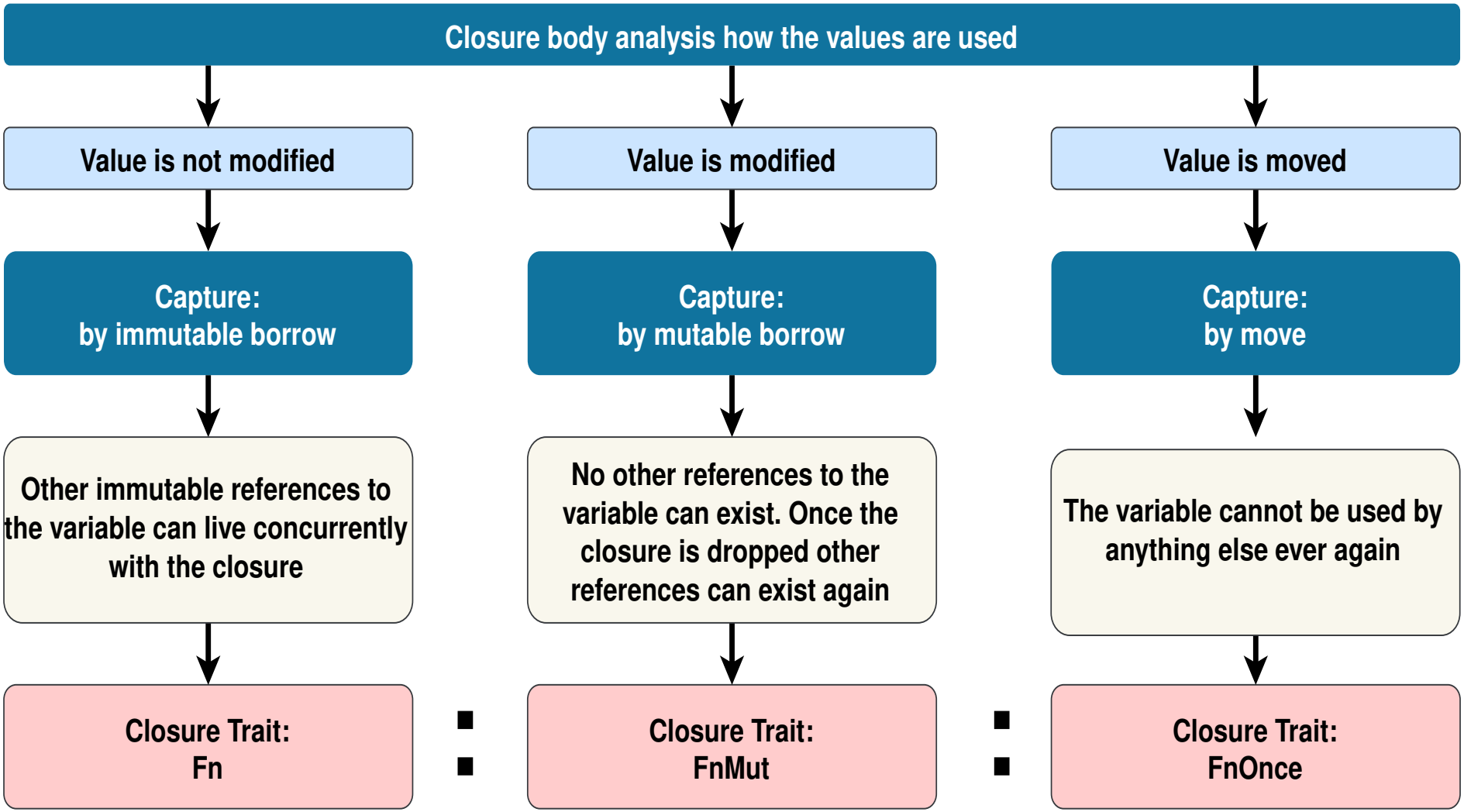
Funciones anónimas (Closures) I

1. FnOnce
2. FnMut.

Funciones anónimas (Closures) I

1. FnOnce
2. FnMut.
3. Fn.

Funciones anónimas (Closures) II



Ejemplos Funciones anónimas (Closures)

```rust
fn main() {
  let x = String::from("hello there !");
  // FnOnce closure si mueve el valor y se pierde
  let consume_and_return_x = move |y| y;
  println!("{}", consume_and_return_x(x));
  // Si la llamas otra vez no compila
  // FnMut closure cambia algun valor
  let square = |x: &mut u8| *x *= *x;
  let mut z: u8 = 5;
  square(&mut z);
  println!("{}", z);
  // Fn closure si no cambia ningun valor
  let n: u8 = 4;
  let mult_by_two = |x: &u8| -> u8 { *x * 2 };
  println!("El doble de {} es {}", n, mult_by_two(&n))
}
```

Colecciones e iteradores con Closures

```
1 pub fn search_case_insensitive<'a>(query: &str,
2 contents: &'a str) -> Vec<&'a str> {
3   // Atencion al iterador lines
4   let query: &str = &(query.to_lowercase());
5   contents
6     .lines()
7     .filter(|line|
8       line.to_lowercase().contains(query))
9     .collect()
10 }
```

Special Traits

1. Display.

Special Traits

1. Display.
2. Debug.

Special Traits

1. Display.
2. Debug.
3. Clone.

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...
6. Sized.

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...
6. Sized.
7. Iter Traits: Iterator, FromIterator, IntoIterator, DoubleEndedIterator...

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...
6. Sized.
7. Iter Traits: Iterator, FromIterator, IntoIterator, DoubleEndedIterator...
8. Comparation Traits : Eq, PartialEq, Ord, PartialOrd.

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...
6. Sized.
7. Iter Traits: Iterator, FromIterator, IntoIterator, DoubleEndedIterator...
8. Comparation Traits : Eq, PartialEq, Ord, PartialOrd.
9. Hash.

Special Traits

1. Display.
2. Debug.
3. Clone.
4. Copy.
5. Operator Traits : Deref, DerefMut, Add, Mul, Fn, FnMut, FnOnce, Drop ...
6. Sized.
7. Iter Traits: Iterator, FromIterator, IntoIterator, DoubleEndedIterator...
8. Comparation Traits : Eq, PartialEq, Ord, PartialOrd.
9. Hash.
10. Future.

Special Traits I

```rust
use std::fmt;
use std::ops::Add;
#[derive(Debug, Copy, Clone, PartialEq, Hash)]
struct Point {
  x: i32,
  y: i32,
}
impl fmt::Display for Point {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "({}, {})", self.x, self.y)
  }
}
```

Special Traits II

```
impl Point {
  fn new(x: i32, y: i32) -> Point {
    Point { x, y }
  }
}
trait OutlinePrint: fmt::Display {
  fn outline_print(&self) {
    let output = self.to_string();
    let len = output.len();
    println!("\t{}", "*".repeat(len + 4));
    println!("\t*{}*", " ".repeat(len + 2));
    println!("\t* {} *", output);
    println!("\t*{}*", " ".repeat(len + 2));
    println!("\t{}", "*".repeat(len + 4));
  }
}
```

Special Traits III

```
impl OutlinePrint for Point {}
// Implementamos el operador +
impl Add for Point {
  type Output = Point;
  fn add(self, other: Point) -> Point {
    Point {
      x: self.x + other.x,
      y: self.y + other.y,
    }
  }
}
```

Special Traits IV

```
fn main() {
  let p: Point = Point::new(32, 6);
  p.outline_print();
  println!("++++++++++++++++++++++++++++++");
  let p2: Point = Point::new(6, 32);
  p2.outline_print();
  println!("==============================");
  let p3: Point = p + p2;
  p3.outline_print();
}
```

Fearless Concurrency

1. Safe Rust elimina los Data Races con el sistema de Borrowing, (ojo puedes seguir teniendo un General Race).

Fearless Concurrency

1. Safe Rust elimina los Data Races con el sistema de Borrowing, (ojo puedes seguir teniendo un General Race).
2. Los hilos pueden devolver valores.

Fearless Concurrency

1. Safe Rust elimina los Data Races con el sistema de Borrowing, (ojo puedes seguir teniendo un General Race).
2. Los hilos pueden devolver valores.
3. Los traits Send y Sync , permiten mandar valores a otros hilos y permiten compartir valores entre hilos respectivamente (Safe Rust).

Fearless Concurrency

1. Safe Rust elimina los Data Races con el sistema de Borrowing, (ojo puedes seguir teniendo un General Race).
2. Los hilos pueden devolver valores.
3. Los traits Send y Sync , permiten mandar valores a otros hilos y permiten compartir valores entre hilos respectivamente (Safe Rust).
4. Structs de Sincronización entre hilos Arc , Mutex , RwLock y Atomic types.

Ejemplo concurrencia

```
1 use std::thread;
2 use std::time::Duration;
3 pub struct Th {
4   number: u32,
5   wait_time: u64,
6 }
7 impl Th {
8   pub fn new(number: u32, wait_time: u64) -> Self {
9     Th { number, wait_time }
10  }
11  pub fn start(self) -> std::thread::JoinHandle<()> {
12    thread::spawn(move || {
13    self.run();
14    })
15  }
```

Ejemplo concurrencia

```
1   pub fn run(&self) {
2     println!(
3     "Thread {} ha empezado y tiene
4     un tiempo de espera {}",self.number, self.wait_time
5     );
6     let time = Duration::from_millis(self.wait_time);
7     thread::sleep(time);
8     println!(
9     "Thread {} ha terminado y tenia
10    un tiempo de espera {}",self.number, self.wait_time
11    );
12  }
13 }
```

Ejemplo concurrencia

```
#![feature(thread_is_running)]
mod primer;
use crate::primer::*;
use rand::{thread_rng, Rng};
use std::thread::JoinHandle;
fn main() {
  let mut rng = thread_rng();
  let n: u32 = rng.gen_range(1..=10000);
  let mut threads: Vec<JoinHandle<()>> = vec![];
  for i in 1..=n {
    let time: u64 = rng.gen_range(0..=3) * 1000;
    let t = Th::new(i, time);
    let t = t.start();
    threads.push(t);
  }
```

Ejemplo concurrencia

```
1   let mut i: usize = 0;
2   while i < threads.len() {
3     let th = threads.swap_remove(0);
4     // Ojo funcion unicamente en Nightly Rust.
5     if th.is_running() {
6       th.join().unwrap();
7     }
8     i = i + 1;
9   }
10  println!("Main Siempre ultimo :D");
11 }// llave del main
```

Smart Pointers I

¿Qué son los Smart Pointers?

Un Smart Pointer es un tipo de dato abstracto que simula ser un puntero mientras que implementa algunas funcionalidades, como la gestión automática de memoria.

Smart Pointers I

¿Qué son los Smart Pointers?

Un Smart Pointer es un tipo de dato abstracto que simula ser un puntero mientras que implementa algunas funcionalidades, como la gestión automática de memoria.

Tipos de Smart Pointers.

1. Box

Smart Pointers I

¿Qué son los Smart Pointers?

Un Smart Pointer es un tipo de dato abstracto que simula ser un puntero mientras que implementa algunas funcionalidades, como la gestión automática de memoria.

Tipos de Smart Pointers.

1. Box
2. Rc (Reference Counting)

Smart Pointers I

¿Qué son los Smart Pointers?

Un Smart Pointer es un tipo de dato abstracto que simula ser un puntero mientras que implementa algunas funcionalidades, como la gestión automática de memoria.

Tipos de Smart Pointers.

1. Box
2. Rc (Reference Counting)
3. Vec

Smart Pointers I

¿Qué son los Smart Pointers?

Un Smart Pointer es un tipo de dato abstracto que simula ser un puntero mientras que implementa algunas funcionalidades, como la gestión automática de memoria.

Tipos de Smart Pointers.

1. Box
2. Rc (Reference Counting)
3. Vec
4. RefCell

Smart Pointer I

```rust
use std::fmt;
#[derive(Debug, PartialEq, Eq)]
pub struct List<T> where T: fmt::Display, {
  head: Link<T>,
  len: usize,
}
type Link<T> = Option<Box<Node<T>>>;
#[derive(Debug, PartialEq, Eq)]
struct Node<T> where T: fmt::Display,{
  elem: T,
  next: Link<T>,
}
```

Smart Pointer II

```
impl<T> fmt::Display for List<T> where T: fmt::Display, {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "[")?;
    let opt = self.head.as_ref();
    match opt {
      None => {
        write!(f, "]")
      }
      Some(boxed) => {
        // Option<&Box<Node<T>>>
        let mut next_opt = (**boxed).next.as_ref();
        if let None = next_opt {
          write!(f, "{}]", **boxed)
        }
```

Smart Pointer II

```
        else {
          write!(f, "{}", **boxed)?;
          while let Some(boxx) = next_opt {
            next_opt = (**boxx).next.as_ref();
            if let None = next_opt {
              write!(f, "{}]", **boxx)?;
            } else {
              write!(f, " -> {}", **boxx)?;
            }
          }
          Ok(())
        }
      }// llave Some
    }//llave match
  }// llave fmt
}// llave impl
```

Unsafe Rust

¿Qué es Unsafe Rust?

Es una parte del lenguaje Rust que permite realizar algunas funcionalidades que no están permitidas en Safe Rust.

Funcionalidades que permite hacer Unsafe Rust.

1. Dereference raw pointers.

Unsafe Rust

¿Qué es Unsafe Rust?

Es una parte del lenguaje Rust que permite realizar algunas funcionalidades que no están permitidas en Safe Rust.

Funcionalidades que permite hacer Unsafe Rust.

1. Dereference raw pointers.
2. Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator).

Unsafe Rust

¿Qué es Unsafe Rust?

Es una parte del lenguaje Rust que permite realizar algunas funcionalidades que no están permitidas en Safe Rust.

Funcionalidades que permite hacer Unsafe Rust.

1. Dereference raw pointers.
2. Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator).
3. Implement unsafe traits (Send and Sync).

Unsafe Rust

¿Qué es Unsafe Rust?

Es una parte del lenguaje Rust que permite realizar algunas funcionalidades que no están permitidas en Safe Rust.

Funcionalidades que permite hacer Unsafe Rust.

1. Dereference raw pointers.
2. Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator).
3. Implement unsafe traits (Send and Sync).
4. Mutate statics.

Unsafe Rust

¿Qué es Unsafe Rust?

Es una parte del lenguaje Rust que permite realizar algunas funcionalidades que no están permitidas en Safe Rust.

Funcionalidades que permite hacer Unsafe Rust.

1. Dereference raw pointers.
2. Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator).
3. Implement unsafe traits (Send and Sync).
4. Mutate statics.
5. Access fields of unions.

Ejemplo Unsafe Rust

```rust
#[derive(Debug, PartialEq, Eq)]
pub struct List<T> {
  head: Link<T>,
  tail: Link<T>,
  len: usize,
}
// Atencion raw pointer
type Link<T> = *mut Node<T>;
#[derive(Debug, PartialEq, Eq)]
struct Node<T> {
  elem: T,
  next: Link<T>,
}
```

Ejemplo Unsafe Rust

```
impl<T> fmt::Display for List<T>
where
T: fmt::Display + Clone,
{
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "[")?;
    unsafe {
      if self.head.is_null() {
        write!(f, "]")
      } else {
        let mut aux = self.head;
        write!(f, "{}", (*aux).elem)?;
        aux = (*aux).next;
        if aux.is_null() {
          write!(f, "]")
        }
```

Ejemplo Unsafe Rust

```
2           else {
3               while !aux.is_null() {
4               write!(f, " -> {}", (*aux).elem)?;
5               aux = (*aux).next;
6               }
7               write!(f, "]")
8           }
9         }// segundo else
10      }// unsafe
11    } // fmt
12 } // impl
```

Macros!

¿Qué son las Macros?

Son la forma de escribir código que escribe otro código. Su principal objetivo es reducir el código drásticamente, generalizar código ya realizado. Usar macros es la forma de hacer Metaprogramming en Rust.

Macros!

¿Qué son las Macros?

Son la forma de escribir código que escribe otro código. Su principal objetivo es reducir el código drásticamente, generalizar código ya realizado. Usar macros es la forma de hacer Metaprogramming en Rust.

Tipos de Macros.

1. Custom #[derive] macros that specify code added with the derive attribute used on structs and enums.

Macros!

¿Qué son las Macros?

Son la forma de escribir código que escribe otro código. Su principal objetivo es reducir el código drásticamente, generalizar código ya realizado. Usar macros es la forma de hacer Metaprogramming en Rust.

Tipos de Macros.

1. Custom #[derive] macros that specify code added with the derive attribute used on structs and enums.
2. Attribute-like macros that define custom attributes usable on any item. (Tienen pinta de funciones).

Macros!

¿Qué son las Macros?

Son la forma de escribir código que escribe otro código. Su principal objetivo es reducir el código drásticamente, generalizar código ya realizado. Usar macros es la forma de hacer Metaprogramming en Rust.

Tipos de Macros.

1. Custom #[derive] macros that specify code added with the derive attribute used on structs and enums.
2. Attribute-like macros that define custom attributes usable on any item. (Tienen pinta de funciones).
3. Function-like macros that look like function calls but operate on the tokens specified as their argument. (Parecen funciones como ejecucion pero tienen argumentos variables)(println! , write! , assert! . . .).

Metavariables en Declarative Macros

Ejemplo Macros!

```
#[path="../../linked_list_second/src/
linked_list_second.rs"]
mod linked_list_second;
use crate::linked_list_second::List;
macro_rules! multiply_add {
    ($a:expr , $b:expr, $c:expr , u16) => {
        ($b + $c) * $a
    };
    ($a:expr , $b:expr, $c:expr , &List<u16>) => {
        (&($b + $c)) * $a
    };
}
```

Ejemplo Macros!

```
fn main() {
  let mut list: List<u16> = List::new();
  let mut list_2: List<u16> = List::new();
  list.append(73);
  list_2.append(1729);
  println!("{}", multiply_add!(2, 3, 4, u16));
  println!(
    "{}",
    multiply_add!(2 + 1, &(&list + &list_2),
      &List::new(), &List<u16>)
  );
}
```

Libros de referencia

1. Rust book
2. The Rustonomicon.
3. Documentación oficial std Rust.
4. Rust Reference
5. Libro Linked List.
6. Diagrama Closures.
7. The Little Book of Rust Macros.
8. Rust Cheat sheet
9. Rust anthology.