



Curso de Rust



1

Antes de Empezar

¿Que espero en este curso?

Qué espero de este curso

- 1 Entender que es Rust
- 2 Tener claro los concetos de *Ownership* y *Borrowing*
- 3 Punteros y referencias
- 4 Que sea lo más practico posible
- 5 Después de cada sección haremos un juego con preguntas sobre lo que hemos visto

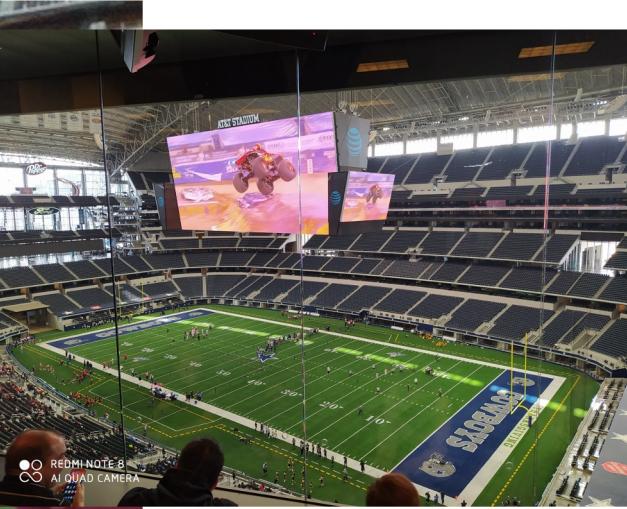


Quien soy



- Ingeniero
 - Informático
 - Teleco
 - industrial
- Matemático
- Doctor en Físicas
- Seguidor de NBA y NFL
- Green bay y Pittsburgh Steelers , odio a los patriots

About me



2

¿Que es Rust?



Historia

Introducción a Rust

- Rust es un lenguaje de programación de sistemas de código abierto.
- Semejante a C++ en su sintaxis.
- Utilizado para construir sistemas operativos, motores de juegos, navegadores y herramientas de línea de comandos.

Historia de Rust



¿Para Qué se Usa Rust?

- Kernels de Sistemas
- Operativos Motores de Juegos
- Motores de Navegadores
- Herramientas de Línea de Comandos

Características de Rust

- Rendimiento
- Seguridad de Memoria
- Concurrency Segura
- Independencia de Plataforma

Rendimiento

- Comparable a C y C++.
- Optimización de bajo nivel.
- Uso eficiente de la memoria.

Seguridad de Memoria en Rust

- Prevención de errores de memoria como desbordamientos de buffer.
- Sistema de tipos que garantiza la seguridad en tiempo de compilación.
- Evita errores comunes que pueden llevar a vulnerabilidades de seguridad.

Concurrencia Segura

- Permite la ejecución paralela sin condiciones de carrera.

- El sistema de préstamos y propiedad asegura que no se puede modificar el mismo dato desde múltiples hilos simultáneamente.

Independencia de Plataforma

- Los programas de Rust pueden ser compilados y ejecutados en diferentes plataformas sin cambios.
- Soporte para una amplia gama de sistemas operativos.

Instalación

Comenzando con Rust

- Usa un compilador en línea para empezar rápidamente.
- Instala Rust en tu PC con `rustup`.

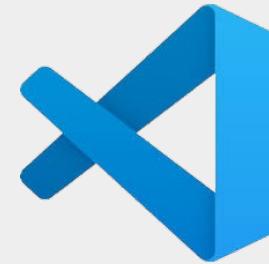
Instalación de Rust

- Windows: Visita rustup.rs y ejecuta rustup-init.exe.
- Unix/Linux/macOS: Usa el terminal con el comando curl

```
# Windows  
rustup.rs  
  
# Unix/Linux/macOS  
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

IDE y plugins

- **CodeLLDB**
- **Crates**
- **Dev Containers**
- **Docker**
- **Error Lens**
- **Even Better TOML**
- **Live Share** ★



Cargo Package Manager for Rust

- Basic Scaffolding
- Dependency Mgmt
- Compilation
- Packaging
- Publishing
- Toolchain Expansion



Primeros Pasos

- Cargo –versión
 - Creación de una aplicación
- Cargo new --bin my-rust-app
 - Creación de una libreria
- Cargo new --lib my-rust-lib
- Cargo new my-rust-app –vcs git



Toml y Lock

- Cargo.toml:

```
[package]  
name = "hello_cargo"  
version = "0.1.0"
```

```
edition = "2021"
```

```
[dependencies]  
rand = "0.8.5"
```



Cargo.lock

Simple rule:

Binary: check Cargo.lock into source control

Library: put Cargo.lock in .gitignore

Don't use "*" dependencies.

Crates.oi Will reject your crate!



Depends On..

Crates from crates.io

```
[dependencies]
chashmap = "2.2.0"
```

Git repos

```
[dependencies]
evmap = { git = "https://github.com/jonhoo/rust-evmap.git" }
```

Files in subdirectories of your project

```
[dependencies]
hello_utils = { path = "hello_utils" }
```



Versions of Truth

Cargo.toml:

```
[dependencies]
clap = "2.20.0"
```

```
>cargo build
Updating crates.io index
...
Compiling clap v2.32.0
```

To ensure specific version -
use:

```
mylib = "2.0.2.3"
```



Versions of Truh : Caret

Se permite una actualización si el nuevo número de versión no modifica el dígito distinto de cero situado más a la izquierda en la agrupación de parches mayores y menores:

```
[dependencies]
time = "^0.20.0"
```

Examples:

```
^1.2.3 := >=1.2.3 <2.0.0
^1.2 := >=1.2.0 <2.0.0
^1 := >=1.0.0 <2.0.0
^0.2.3 := >=0.2.3 <0.3.0
^0.2 := >= 0.2.0 < 0.3.0
^0.0.3 := >=0.0.3 <0.0.4
^0.0 := >=0.0.0 <0.1.0
^0 := >=0.0.0 <1.0.0
```



Versions of Truh : Tilde

Se puede especificar versiones mínimas para cuando se actualice el proyecto

Examples:

```
[dependencies]
clap = "~2.3.0"
```

```
~1.2.3 := >=1.2.3 <1.3.0
~1.2 := >=1.2.0 <1.3.0
~1 := >=1.0.0 <2.0.0
```



The background features a large white circle centered on the page. To its left is a smaller gray circle with a white outline. To its right is a series of concentric white circles on a black background.

3

Conceptos fundamentales

Fundamentos

Variables, inmutabilidad, y constantes

- Una variable es un contenedor que almacena un valor que puede ser utilizado y modificado a lo largo del código.
- En Rust, las variables son **inmutables** por defecto, lo que significa que no pueden ser cambiadas después de ser inicializadas.

Fundamentos

Variables, inmutabilidad, y constantes

- Para declarar una variable en Rust, usamos la palabra clave let seguida del nombre de la variable y opcionalmente su tipo y valor.
- `let x = 5; let y: f64 = 3.5;`
- Características de las Variables:
 - Inmutabilidad por Defecto: Ayuda a prevenir errores relacionados con la mutación accidental de datos.
 - Tipo Inferido: Rust puede inferir el tipo de la variable basado en el valor asignado.

Fundamentos

Variables, inmutabilidad, y constantes

- Shadowing Permitido: Es posible redeclarar una variable con el mismo nombre en el mismo ámbito, una práctica conocida como Shadowing.
- Ámbito y Vida de una Variable:
 - Ámbito: El ámbito de una variable es la región del código donde la variable es válida.
 - Vida de una Variable: Rust usa el sistema de alcance para determinar la vida útil de una variable, que dura hasta que sale de su bloque de código.

Fundamentos

Tipos de datos

```
fn main() {
    let x = 5; // x es válida desde aquí
    {
        let y = 10; // y es válida solo dentro de este bloque
        println!("x: {}, y: {}", x, y);
    } // y deja de ser válida aquí
    println!("x: {}", x);
}
// x deja de ser válida aquí
```

	Tipos	Literales
Enteros con signo	i8, i16, i32, i64, i128, isize	-10, 0, 1_000, 123_i64
Enteros sin signo	u8, u16, u32, u64, u128, usize	0, 123, 10_u16
Números de coma flotante	f32, f64	3.14, -10.0e20, 2_f32
Valores escalares Unicode	char	'a', 'a', '∞'
Booleanos	bool	true, false

Array vs Tuplas



Array:

- **Homogeneidad:** Todos los elementos en un array deben ser del mismo tipo.
- **Tamaño fijo:** El tamaño de un array es fijo y conocido en tiempo de compilación.
- **Sintaxis:** Se definen usando corchetes [] y su tipo incluye el número de elementos.
- **Acceso a elementos:** Se accede a los elementos por índice.
- **Tuplas:**
 - **Heterogeneidad:** Los elementos en una tupla pueden ser de diferentes tipos.
 - **Tamaño fijo:** El tamaño de una tupla también es fijo, pero se define por los tipos de sus elementos, no por un número específico.
 - **Sintaxis:** Se definen usando paréntesis ().
 - **Acceso a elementos:** Se accede a los elementos por posición, usando el operador de punto . seguido del índice.

Ejercicio 1

En este ejercicio, el objetivo es comprender cómo funciona la inmutabilidad de las variables en Rust y cómo se puede usar la palabra clave **mut** para permitir cambios en las variables.

1. Declara una variable `x` de tipo `i32` con el valor `10`.
2. Intenta cambiar el valor de `x` a `20` y observa el error.
3. Ahora, declara una nueva variable `y` usando la palabra clave `mut` y asignale el valor `30`.
4. Cambia el valor de `y` a `40` y imprime ambos valores.

Ejercicio 2

Este ejercicio se centra en la definición y uso de constantes en Rust.

A diferencia de las variables, las constantes deben tener un tipo de dato especificado y su valor no puede cambiar una vez definido.

1. Define una constante **PI** de tipo **f64** con el valor **3.141592**.
2. Imprime el valor de **PI**.
3. Intenta cambiar el valor de **PI** y observa el error.
4. Define una variable mutable **radio** de tipo **f64** con el valor **5.0**.
5. Calcula el área de un círculo usando **PI** y **radio** y asigna el resultado a una nueva variable **área**.
6. Imprime el área del círculo.

Funciones

Las funciones las declaramos como fn <name>

Usamos snake_case

El Orden no es importante al ser compilado

Se puede devolver un valor (expresión) y devuelve el ultimo valor del scope

Variable



Operadores & y * en Rust

```
let x = 5;
let y = &x; // `&` crea una referencia a `x`

println!("Valor de x: {}", x);
println!("Valor de y (referencia a x): {}", y);

let z = *y; // `*` desreferencia y obtiene el valor de `y`

println!("Valor de z (desreferenciado): {}", z);
```

Variable

Paso de Valores por Valor vs. Referencia

```
fn main() {  
    let s1 = String::from("hola");  
    let s2 = s1; // Movimiento (move) por valor  
  
    // let s3 = s1; // Esto causaría un error porque `s1` ya no es válido  
  
    let s4 = String::from("mundo");  
    let s5 = &s4; // Paso por referencia  
  
    println!("s4: {}", s4);  
    println!("s5: {}", s5); // `s5` sigue siendo válido  
}
```

variable-scope

el ámbito de una variable define la región en la que la variable está disponible para su uso

```
fn main() {  
    let age = 31;  
    println!("{}", age)  
}
```

} Todo el main

```
fn main() {  
    let outer_var = 100;  
  
    {  
        let inner_var = 200;  
        println!("inner_var = {}", inner_var);  
    }  
  
    println!("inner_var = {}", inner_var);  
    println!("outer_var = {}", outer_var);  
}
```



¿La variable `inner_var` saldrá por el `println`?

variable-scope

el ámbito de una variable define la región en la que la variable está disponible para su uso

```
fn main() {  
    let outer_var = 100;  
  
    {  
        let inner_var = 200;  
        println!("inner_var = {}", inner_var);  
  
        println!("inner_var = {}", inner_var);  
        println!("outer_var = {}", outer_var);  
    }  
  
    println!("inner_var = {}", inner_var);  
    ^^^^^^^^^ help: a local variable with a similar name exists: `outer_var`
```



variable-scope

el ámbito de una variable define la región en la que la variable está disponible para su uso

```
fn main() {  
    let outer_var = 100;  
  
    {  
        let inner_var = 200;  
        println!("inner_var = {}", inner_var);  
  
        println!("inner_var = {}", inner_var);  
        println!("outer_var = {}", outer_var);  
    }  
  
    println!("inner_var = {}", inner_var);  
    ^^^^^^^^^ help: a local variable with a similar name exists: `outer_var`
```



variable-scope

el ámbito de una variable define la región en la que la variable está disponible para su uso

```
fn main() {  
    let outer_var = 100;  
  
    {  
        let inner_var = 200;  
        println!("inner_var = {}", inner_var);  
        println!("outer_var inside inner block = {}", outer_var);  
    }  
  
    println!("outer_var = {}", outer_var);  
}
```



variable-scope

```
fn main() {  
    let outer_var = 100;  
  
    {  
        let inner_var = 200;  
  
        println!("inner_var = {}", inner_var);  
        println!("outer_var inside inner block = {}", outer_var);  
    }  
  
    println!("outer_var = {}", outer_var);  
}
```

Scope of
inner_var
variable

Scope of
outer_var
variable

variable-Shadowing

En Rust, cuando una variable declarada dentro de un ámbito particular tiene el mismo nombre que una variable declarada en el ámbito externo, se conoce como variable shadowing.

Podemos utilizar el mismo nombre de variable en diferentes bloques de ámbito en el mismo programa.

```
fn main() {  
    let random = 100;  
  
    {  
        println!("random variable before shadowing in inner block = {}", random);  
        let random = "abc";  
        println!("random after shadowing in inner block = {}", random);  
    }  
  
    println!("random variable in outer block = {}", random);  
}
```

Uso de los Operadores & y *

En Rust, cuando una variable declarada dentro de un ámbito particular tiene el mismo nombre que una variable declarada en el ámbito externo, se conoce como variable shadowing.

Podemos utilizar el mismo nombre de variable en diferentes bloques de ámbito en el mismo programa.

```
fn main() {  
    let random = 100;  
  
    {  
        println!("random variable before shadowing in inner block = {}", random);  
        let random = "abc";  
        println!("random after shadowing in inner block = {}", random);  
    }  
  
    println!("random variable in outer block = {}", random);  
}
```

Fundamentos

Tipos de datos (**String**) &**str** (**String Slice**)

- Referencia a una secuencia de caracteres en otro lugar, como en el literal de una cadena.
- Características:
 - Inmutable.
 - Generalmente usado para cadenas que no necesitan ser modificadas.

```
let greeting: &str = "Hello, world!";
```

Fundamentos

Tipos de datos (String)

- Cadena de caracteres que se almacena en el heap, permitiendo que sea mutable y redimensionable.
- Características:
 - Mutable.
 - Propiedad exclusiva: se puede transferir la propiedad de la cadena.

Fundamentos

Tipos de datos (String)

- Conclusión
 - Rust ofrece un manejo robusto de cadenas de caracteres a través de &str y String, combinando seguridad y flexibilidad.
 - Entender las peculiaridades de cada tipo es crucial para evitar errores comunes y optimizar el rendimiento.

Fundamentos

gestión de referencias (String)

- `&str` es una referencia a una cadena de caracteres (`str`) en Rust, y se usa comúnmente para representar literales de cadena o fragmentos de cadena sin necesidad de asignar memoria adicional.
 - `&str`: Es una referencia a una secuencia de caracteres de tamaño fijo en el tiempo de compilación. No se puede modificar y no se asigna memoria adicional.
 - `String`: Es una cadena de caracteres en el heap que es mutable y se puede cambiar en tiempo de ejecución.

Fundamentos

gestión de referencias (String)

- `&str` es una referencia a una cadena inmutable.
- Es eficiente y no requiere asignación adicional de memoria.
- Las referencias tienen vidas útiles que deben ser gestionadas cuidadosamente para evitar errores de memoria.
- Convertir entre `&str` y `String` es común y sencillo.

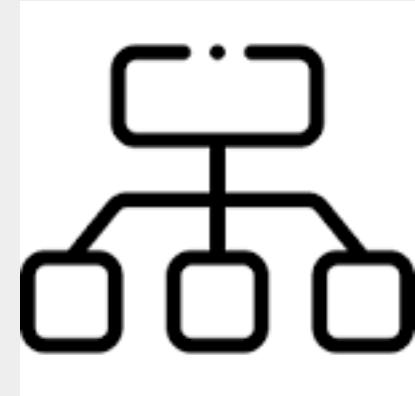
“



Structs

¿Qué es una Struct? Y ¿Por qué usar Structs?

- Estructuras o structs son tipos de datos definidos por el usuario.
- Permiten almacenar datos de diferentes tipos en una sola unidad
- Agrupación lógica de datos relacionados.
- Facilita la manipulación y gestión de múltiples propiedades.



Structs

Definiendo

```
struct NombreStruct {  
    campo1: tipo_de_dato,  
    campo2: tipo_de_dato,  
    campo3: tipo_de_dato,  
}
```

```
struct Persona {  
    nombre: String,  
    edad: u8,  
    altura: u8,  
}
```

Structs

Instanciando

- La instanciación de structs implica asignar valores a cada campo.
- Rust utiliza : para separar el nombre del campo y el valor asignado.

```
let personal1 = Persona {  
    nombre: String::from("John Doe"),  
    edad: 18,  
    altura: 178,  
};
```

Structs

Accediendo

- La instanciación de structs implica asignar valores a cada campo.
- Rust utiliza : para separar el nombre del campo y el valor asignado.

```
println!("Nombre: {}", persona1.nombre);
println!("Edad: {}", persona1.edad);
println!("Altura: {}", persona1.altura);
```

Structs

Desestructuración

- Desestructuración permite extraer los campos de una struct en variables individuales.
- Útil para manipulación directa y fácil acceso a los datos.

```
let Persona { nombre, edad, altura } = personal;
```

Ejercicio 3

1. Define una struct llamada Vehiculo con los campos marca (String), modelo (String) y anio (u16).
2. Crea una instancia de Vehiculo y asigna valores a sus campos.
3. Imprime los valores de los campos del Vehiculo.

Ejercicio 4

Tarea1: Define una struct Estudiante con campos nombre, edad y nota.

Objetivo: Instanciar la struct con datos de un estudiante y acceder a sus campos.

Tarea2: Crea una struct Producto con campos nombre, precio y cantidad. Desestructura una instancia y modifica los valores.

Objetivo: Familiarizarse con la desestructuración y la manipulación de structs.

Tarea3: Define una struct Direccion con campos calle, ciudad y pais. Luego, crea una struct Persona que incluya una instancia de Direccion.

Objetivo: Entender cómo trabajar con structs anidadas.

“



Enums

Cuándo Usar Enums

- Un tipo de dato definido por el usuario.
- Permite seleccionar un valor de una lista de valores relacionados.
- Almacenar un conjunto de direcciones conocidas.

```
let my_sport = Sport::Football;
```

```
enum Sport {  
    Basketball,  
    Volleyball,  
    Football,  
    Cricket,  
}
```

Enums

Convenciones de Nomenclatura

- Nombre de la enum y sus variantes con la primera letra en mayúscula.

Accediendo a Variantes de Enums

```
let my_sport = Sport::Football;
```

Enums

Enums con Valores Asociados

```
#[derive(Debug)]
enum Result {
    Score(f64),
    Valid(bool),
}
```

```
let num = Result::Score(3.14);
println!("{:?}", num);
```

```
num = Score(3.14)
```

Enums

Conclusión

- Enums permiten manejar valores relacionados.
- Soportan diferentes tipos de datos.
- Pueden ser mutables para mayor flexibilidad

Enums

Tipos genéricos

- Imaginemos que tenemos una variable nombre y que en algún momento puede estar nula, Rust intenta evitar esas acciones para evitar errores, como el típica java null pointer Exception

```
// enum de Option es tan útil que está inclu
// eliminar
enum Option<T>{
    Some(T),
    None,
}

fn main(){
    let nombre : Option<String> = None;
    println!("{}",nombre)

}
```

Enums

Tipos genéricos

```
#[allow(dead_code)]
// enum de Option es tan útil que está inclu
// eliminar

fn main(){
    let nombre : Option<String> = None;
    nombre = Some("Rafa".to_string());
    //nos da un error por que necesitamos
    // que sea un string
    match nombre {
        None => println!("nombre no valido")
        Some(nombre) =>println!("{}",nombre)
    }
}
```

```
let mut nombre : Option<String> = None;

let nombre : Option<String> = Some("Rafa".to_string());
```

Tenemos que indicar que es mutable añadiendo `mut` a la variable (v1) o inicializamos (v2)

Enums

Tipos genéricos

Vamos con otra manera

- Definimos una struct Usuario con nombre y edad (este valor puede que este o no)

```
fn main(){
    let nuevo_usuario = Usuario{
        name: "Rafa".to_string(),
        age: 12,
    };

    let edad_metodo = nuevo_usuario.getAge();
    println!("Edad: {}", edad_metodo)
}

/* puede pasar que age tenga o no valor*/
struct Usuario{
    name:String,
    age:i32,
}

impl Usuario {
    fn getAge(&self) -> i32{
        self.age
    }
}
```

Enums

Tipos genéricos

```
fn main() {
    // Creación de un nuevo usuario con un valor opcional para la edad
    let nuevo_usuario = Usuario {
        name: "Rafa".to_string(),
        // Utilizamos Some(12) para representar una edad opcional
        age: Some(12),
    };

    // Obtenemos la edad utilizando el método getAge
    let edad_método = nuevo_usuario.getAge();

    // Utilizamos un match para manejar el valor de la opción
    match edad_método {
        Some(age) => println!("Edad: {}", age), // Desempaquetamos y mostramos la edad si
        None => println!("Edad: No especificada"), // Mensaje alternativo si no hay edad
    }
}

// Definición de la estructura Usuario
struct Usuario {
    name: String,
    // La edad es una opción que puede o no contener un valor
    age: Option<i32>,
}

// Implementación de métodos para la estructura Usuario
impl Usuario {
    // Método para obtener la edad, retorna una opción
    fn getAge(&self) -> Option<i32> {
        self.age
    }
}
```

Generics

Tipos genéricos

- Que hacer cuando no quieres definir el tipo de Compilación, para ellos Rust tiene tipos genéricos como en otros lenguajes.

Generics

Tipos genericos

- Vamos a definir un punto (x e y):

```
struct Point {  
    x:i32,  
    y:i32,  
}
```



```
fn main(){  
    let point = Point{  
        x:0,  
        y:012  
    }  
}
```

- En esta definición siempre serán enteros, pero si necesitamos mas precision

Generics

Tipos genéricos

- Necesitamos definir la “Estructura” con tipo Genérico

```
strunct PointGeneric<T> {  
    x:T  
    y:T  
}
```



```
let point3 = PointGenericv2 {  
    x:5,  
    y:12.0,  
}
```

```
strunct PointGenericv<T,V> {  
    x:T  
    y:V  
}
```



```
let point3 = PointGenericv2 {  
    x:5,  
    y:12.0,  
}
```

Ejercicio 5

1. Define un enum **Estado** con variantes para **Activo**, **Inactivo** y **Pendiente**.

TIP: Usa enums cuando una variable pueda tener varios estados posibles, y quieras escribir código específico para cada estado.

2. Define un enum **Mensaje** con variantes para **Texto(String)**, **Numero(i32)** y **Estado(Estado)**.

1. Usa enums con datos asociados para encapsular diferentes tipos de datos dentro de una misma estructura.

3. Define un enum **Resultado** con variantes para **Exito(Estudiante)** y **Error(String)**. Usa la struct **Estudiante** del ejercicio anterior.

1. Usa enums para representar el resultado de operaciones que pueden tener éxito o fallar, y usa structs para encapsular datos complejos.

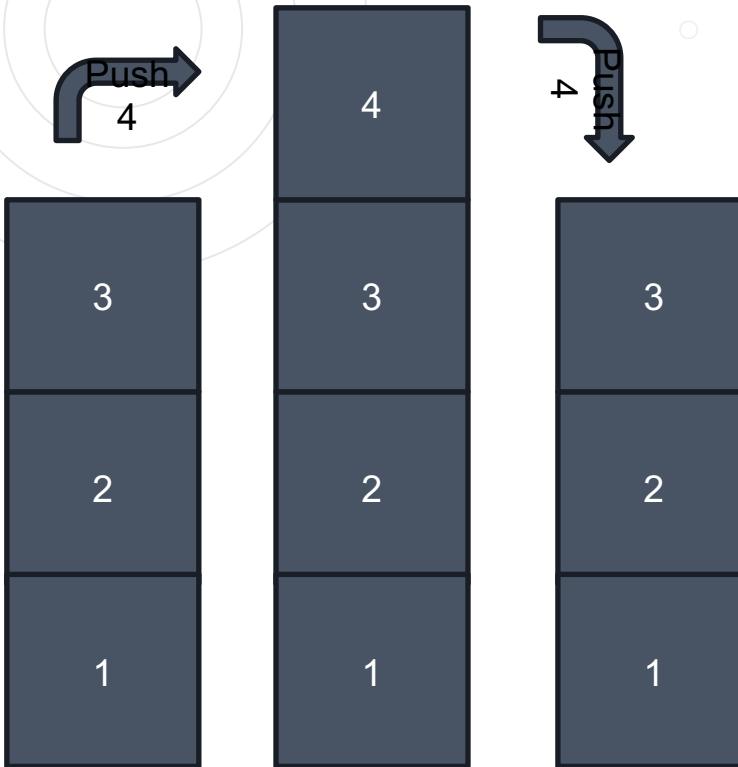
“



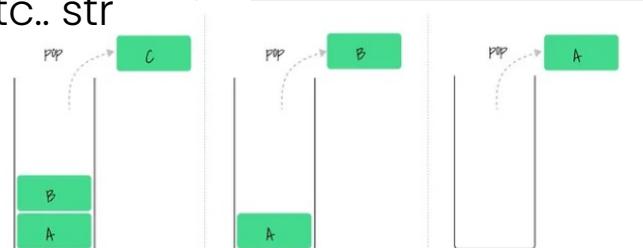
Stack

Time Complexity: $O(1)$ (constant time).

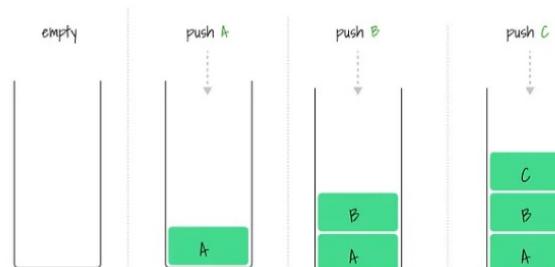
Space Complexity: $O(1)$ (constant space per element).



- Stack : Lo usa para almacenar información
- Al momento de hacer push se guarda arriba de la pila y es muy efectivo
- Ha de tener tamaño conocido : numero i32, f64 , bool, etc.. str



Stack : Pop Operation



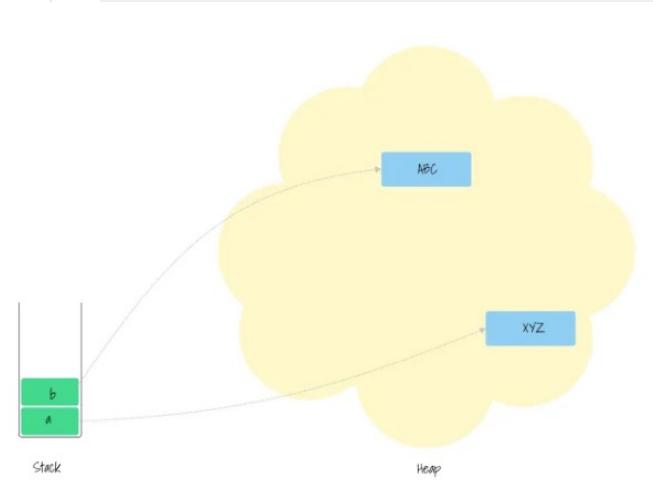
Stack : push operation

Heap

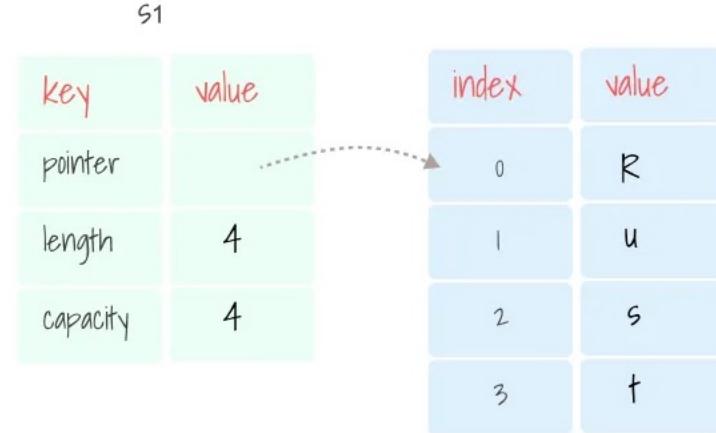
$O(1)$ to $O(\log n)$ or even $O(n)$

$O(1)$

- Heap : Es un poco mas desordenado y lo que hace es que busca memoria libre para guardar información y lo hace sin orden, es mas lento.
- Eso si podemos aumentar tamaño, String, vectores, etc..
- Todo aquello que puede ampliar información.



- `let s1 = String::from("Rust");`
- *Se guarda en el heap no en la pila*



- creamos dos variables :
 - Saludo y saludo1

```
let saludo = "Hola mundo";
let saludo1 = String::from("Hola
mundo");
```

s1

becomes invalid

key	value
pointer	
length	4
capacity	4

s2

key	value
pointer	
length	4
capacity	4

stack

index	value
0	R
1	u
2	s
3	t

heap

- Vamos a imprimirlos
- Vamos a sacar su longitud `.len()`
- Vamos a sacar su capacidad `.capacity()`
- Nos va a dar un error en
 - `println!("{}", saludo.capacity());`
- ¿Por qué?

s1

becomes invalid

key	value
pointer	
length	4
capacity	4

s2

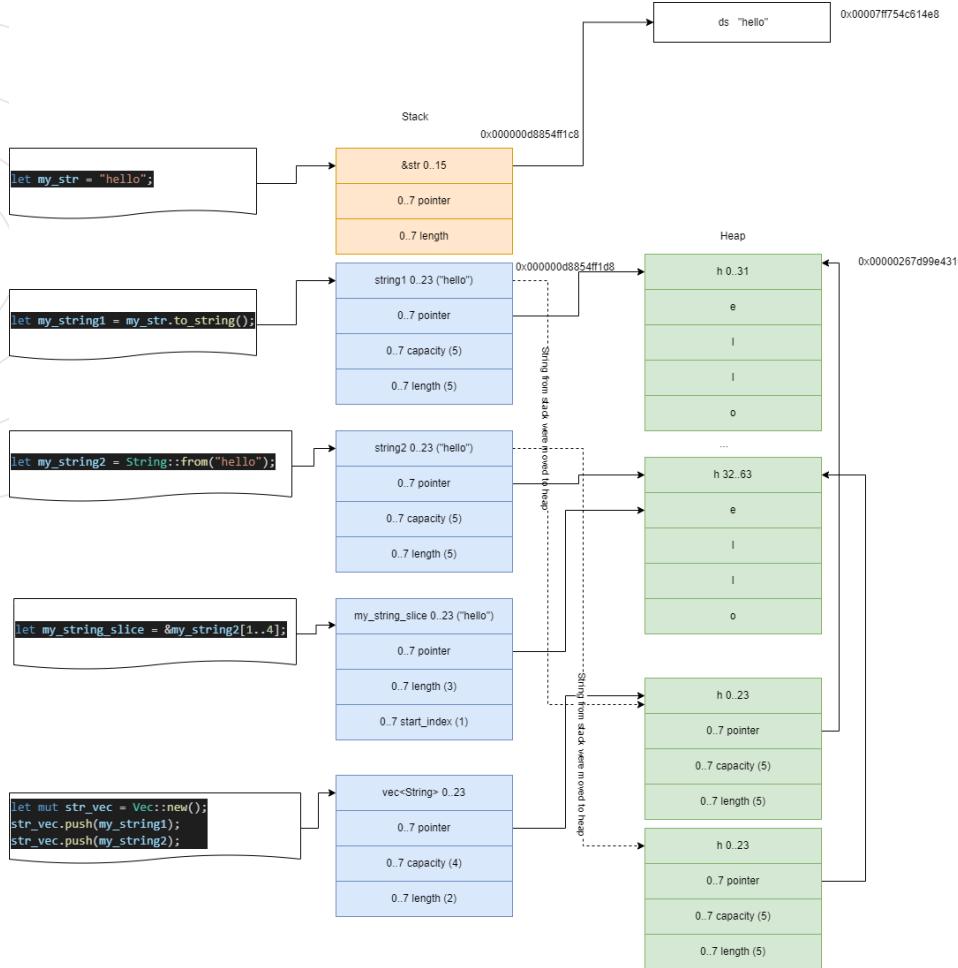
key	value
pointer	
length	4
capacity	4

stack

index	value
0	R
1	u
2	s
3	t

heap

- Vamos a intentar concatenar a mi string "agur Rust" .*push_str*(" agur Rust ");



Referencias y punteros

- Ejemplos

Traits

Introducción

- Un Trait en Rust define funcionalidad compartida para múltiples tipos.
- **Objetivo:** Promover la seguridad de tipos y prevenir errores en tiempo de compilación.
- **Comparación:** Actúan como interfaces en otros lenguajes, pero con algunas diferencias específicas.

Traits

Definición

```
trait NombreTrait {  
    fn metodo_uno(&self, [argumentos: tipo_argumento]) -> tipo_retorno;  
    fn metodo_dos(&mut self, [argumentos: tipo_argumento]) -> tipo_retorno;  
}
```

- **NombreTrait:** Nombre del trait
- **.metodo_uno** y **metodo_dos:** Nombres de los métodos en el trait.
- **&self** y **&mut self:** Referencias al valor self, mutable o no.
- **[argumentos: tipo_argumento]:** Lista de argumentos (opcional).
- **tipo_retorno:** Tipo que retorna el método.

Traits

Generación de Ejemplo

- Vamos a montar dos struct (Gato y Humano).
- Implementamos el trait Hablar
- Añadimos el impl de di_hola e idioma

Trait Debug y atributo Derive

- `PrintIn` o `Print` es una macro (a grandes rasgos es un código que genera otro código).
 - Las macros lo veremos en un apartado específico y construiremos unas macros.
- Podemos usar trait para añadir métodos o funciones
- Atributo Derive nos ayuda a generar implementaciones sobre traits de forma automática como `Copy`, `Clone` o `Derive`

Trait Display

- El trait Display permite personalizar cómo se muestra un tipo de dato como una cadena.
- Es útil para formatear la salida de una manera legible para los usuarios.

```
use std::fmt;

struct User {
    nombre: String,
    edad: i32,
}

impl fmt::Display for User {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{} ({})", self.nombre, self.edad)
    }
}
```

Trait Display

- Se implementa el trait Display utilizando el método fmt.
 - fmt define la lógica de cómo se debe formatear el tipo de dato.
 - fmt::Result indica si la operación de formateo fue exitosa o no.

Trait Display Uso

```
fn main() {  
    let user = User {  
        nombre: "Alice".to_string(),  
        edad: 30,  
    };  
  
    println!("{}", user); // Salida: Alice (30)  
}
```

Ejemplo Strunct

- `struct User{`
- `nombre: String,`
- `edad: i32,`
- `}`

Trait Default

- El trait Default permite especificar valores predeterminados para una estructura.
- Es útil cuando queremos inicializar estructuras con valores por defecto.

```
struct Config {  
    nombre: String,  
    edad: i32,  
}  
  
impl Default for Config {  
    fn default() -> Self {  
        Config {  
            nombre: String::from("Desconocido"),  
            edad: 0,  
        }  
    }  
}
```

Trait Implementación

- Se implementa Default definiendo el método default.
- default retorna una instancia de la estructura con valores predeterminados.
- Se crea una instancia usando Default::default().

```
fn main() {  
    let config: Config = Default::default();  
    println!("Nombre: {}, Edad: {}", config.nombre, config.edad);  
    // Salida: Nombre: Desconocido, Edad: 0  
}
```

Trait

Display vs Default

- *Display*
 - Personaliza la representación como cadena de un tipo.
 - Útil para mostrar datos de manera legible.
- *Default*
 - Define valores predeterminados para una estructura.
 - Facilita la creación de instancias con valores por defecto.

“



Ejercicio 6

1. Define un trait `Describable` con un método `descripcion` que devuelva una cadena (`&str`). Implementa este trait para la struct `Producto`.
 1. TIP: Los traits son similares a las interfaces en otros lenguajes de programación. Permiten definir comportamientos que pueden ser implementados por diferentes tipos.
2. Define un trait `EstadoInfo` con un método `estado_actual` que devuelva una cadena (`&str`). Implementa este trait para el enum `Estado`.
 1. TIP: Cuando implementas un trait para un enum, puedes usar una sentencia `match` para definir el comportamiento de cada variante del enum.
3. Define un trait `DetallesPedido` con un método `detalles` que devuelva una cadena (`String`). Implementa este trait para el enum `Pedido`, que usa la struct `Producto`.
 1. TIP: Implementar traits para enums que contienen structs puede ayudarte a manejar estructuras de datos complejas de manera organizada y con un comportamiento bien definido.

Iteradores

Introducción

- Un iterador en Rust es responsable de crear una secuencia de valores.
- Nos permite iterar sobre cada ítem de la secuencia.
- Principalmente utilizado para bucles.

Iteradores

Iterando Sobre un Arreglo Ejemplo

- Un iterador en Rust es responsable de crear una secuencia de valores.
- Nos permite iterar sobre cada ítem de la secuencia.
- Principalmente utilizado para bucles.

```
let numbers = [2, 1, 17, 99, 34, 56];
let numbers_iterator = numbers.iter();

for number in numbers_iterator {
    println!("{}", number);
}
```

Iteradores

next() del Iterador

```
let colors = vec!["Rojo", "Amarillo", "Verde"];
let mut colors_iterator = colors.iter();

println!("{:?}", colors_iterator.next());
println!("{:?}", colors_iterator.next());
println!("{:?}", colors_iterator.next());
println!("{:?}", colors_iterator.next());
```

Iteradores

Formas de Crear un Iterador

- Usando `iter()`: Pide prestados (referencia) los elementos.
- Usando `into_iter()`: Mueve los elementos.
- Usando `iter_mut()`: Pide prestados y permite modificar los elementos.

Iteradores

Formas de Crear un Iterador

- Usando `iter()`: Pide prestados (referencia) los elementos.

```
let colors = vec!["Rojo", "Amarillo", "Verde"];  
  
for color in colors.iter() {  
    println!("{}", color);  
}  
println!("colors = {:?}", colors);
```

Iteradores

Formas de Crear un Iterador

- Usando `into_iter()`: Mueve los elementos.

```
let mut colors = vec!["Rojo", "Amarillo", "Verde"];  
  
for color in colors.iter_mut() {  
    *color = "Negro";  
    println!("{}", color);  
}  
println!("colors = {:?}", colors);
```

Iteradores

Formas de Crear un Iterador

- Usando `iter_mut()`: Pide prestados y permite modificar los elementos.

```
let mut colors = vec!["Rojo", "Amarillo", "Verde"];  
  
for color in colors.iter_mut() {  
    *color = "Negro";  
    println!("{}!", color);  
}  
println!("colors = {:?}", colors);
```

Iteradores

Adaptadores de Iteradores

- Los adaptadores de iteradores transforman iteradores.
- Ejemplo: `map()`

```
let numbers = vec![1, 2, 3];
let even_numbers: Vec<i32> = numbers.iter().map(|i| i * 2).collect();

println!("numbers = {:?}", numbers);
println!("even_numbers = {:?}", even_numbers);
```

Iteradores

Rango en Rust

```
for i in 1..6 {  
    println!("{}", i);  
}
```

Iteradores

Conclusión

- Los iteradores son una poderosa herramienta en Rust.
- Facilitan el manejo y la transformación de colecciones de datos.
- Diferentes métodos (`iter()`, `into_iter()`, `iter_mut()`) proporcionan flexibilidad en cómo interactuamos con las colecciones.

“



Ejercicio 7

Parte 1: Escribe un programa en Rust que cree un vector de números enteros y utilice un iterador para sumar todos sus elementos. El programa debe imprimir la suma total.

Parte 2 : Crea un programa en Rust que genere un vector de números enteros del 1 al 10. Utiliza un iterador para filtrar los números pares y almacénalos en un nuevo vector. Finalmente, imprime el vector resultante.

Parte 3 : Escribe un programa en Rust que tome un vector de números enteros y utilice el método map para incrementar cada número en 1. Luego, imprime el vector resultante.

Closures

Introducción

1. Los closures son funciones anónimas que se pueden almacenar en variables y pasar como argumentos a otras funciones.
2. Son similares a las lambdas en otros lenguajes de programación.
3. Permiten capturar variables del entorno donde se definen.
4. El compilador de Rust puede inferir los tipos de los parámetros y el tipo de retorno de los closures.
5. No es necesario especificar explícitamente los tipos.

```
let add = |a, b| a + b;  
println!("Suma: {}", add(2, 3)); // Salida: Suma: 5
```

```
let closure_name = |param1, param2| {  
    // cuerpo del closure  
};
```

Closures capturando variables

```
let x = 10;
let add_x = |y| y + x;
println!("Resultado: {}", add_x(5)); // Salida: Resultado: 15
```

- Por referencia: &T
- Por valor: T
- Por mutabilidad: &mut T

Closures

closures y propiedad

```
let x = String::from("Hola");
let consume_x = || drop(x);
consume_x();
// println!("{}", x); // Esto daría error porque `x` ha sido consumido
```

- Los closures pueden mover o copiar variables del entorno.
- Una vez que un closure ha capturado una variable por valor, esa variable ya no se puede usar en el contexto original.

Closures

closures y propiedad

- Los closures se pueden pasar como argumentos a otras funciones.
- Se utilizan comúnmente en funciones de alto orden como map, filter, y for_each.
- Los closures pueden capturar su entorno, mientras que las funciones regulares no.
- Las funciones tienen firmas fijas, mientras que los closures pueden inferir tipos y parámetros.

Ejercicio 8

- Escribe un closure que tome dos números enteros y devuelva su suma. Luego, llama al closure con los números 3 y 4, e imprime el resultado.

“ ”



Repaso Ejercicios

Vamos a simular una pequeña base de datos de empleados utilizando structs y enums para manejar el estado civil de cada empleado.

1. Define una estructura Empleado que contenga los siguientes campos:
 1. nombre: String
 2. edad: u32
 3. estado_civil: Enumeración EstadoCivil con los siguientes casos:
 1. Soltero
 2. Casado
 3. Divorciado
 4. Viudo
2. Implementa una función nuevo_empleado que cree y devuelva un Empleado con los valores proporcionados como parámetros.

Repaso Ejerciceos

1. Define un vector de Empleado con al menos 5 empleados diferentes.
2. Implementa una función buscar_empleado_por_nombre que tome el nombre de un empleado como parámetro y devuelva un Option<&Empleado> con el empleado encontrado o None si no se encuentra ningún empleado con ese nombre.
3. En una función main, utiliza las funciones definidas para crear empleados, almacenarlos en el vector, y buscar empleados por nombre, mostrando la información correspondiente por consola.

Repaso Ejercicios

- Define un array de números enteros con al menos 10 elementos.
- Utiliza un iterador para calcular la suma de todos los números en el array.
- Utiliza un iterador para encontrar el número máximo y mínimo en el array.
- Implementa una función filtrar_numeros que tome como parámetro un número entero umbral y devuelva un vector con los números del array que sean mayores que el umbral. Utiliza una closure para realizar la comparación.
- En la función main, utiliza el array definido, llama a las funciones implementadas para calcular la suma, encontrar el máximo y mínimo, y filtrar números según un umbral especificado, mostrando los resultados por consola.

Prelude

- El preludio es la lista de cosas que el Rust importa automáticamente en cada programa del Rust . Se mantiene lo más pequeño posible y se centra en cosas, particularmente en rasgos, que se utilizan en casi todos los programas Rust

Flujos de control

Los básicos de todos los lenguajes:

- If-else
- Loop
- While
- for

Flujos de control

Uso de if let y while let en Rust

if let es una forma concisa de manejar coincidencias en patrones cuando solo te interesa un caso específico.

Permite evitar el uso del bloque completo de match para simplificar el código.

Es especialmente útil cuando trabajas con opciones (Option) o resultados (Result).

```
let some_option = Some(5);
if let Some(value) = some_option {
    println!("El valor es: {}", value);
}
```

Flujos de control

Ejemplo if-let

```
let valor = Some(10);
if let Some(v) = valor {
    println!("Tenemos un valor: {}", v);
} else {
    println!("No hay valor.");
}
```



```
let resultado: Result<i32, &str> = Ok(5);
if let Ok(valor) = resultado {
    println!("Resultado exitoso: {}", valor);
} else {
    println!("Hubo un error.");
}
```

Flujos de control

Ejemplo if-let

```
let edad :Option <i32> = Some(20);
match edad {
    Some(value) => println!("edad: {}", value),
    _=>(),
}
```

```
if let Some(value) = edad {
    println!("edad: {}", value);
}
```



Flujos de control

while let en Rust

while let es una forma de ejecutar un bucle mientras una expresión coincide con un patrón.

Similar a if let, pero continúa ejecutándose mientras la condición sea verdadera.

```
let mut opciones = vec![Some(1), None, Some(2), Some(3), None];
while let Some(Some(valor)) = opciones.pop() {
    println!("Valor extraído: {}", valor);
}
```

Flujos de control

while let en Rust

```
let mut sms = Some(10);
loop {
    match sms {
        Some(value) => {
            if value > 0 {
                println!("algo");
                sms = Some(value -1);

            } else {
                println!("no hace algo");
                sms = None;
            }
        }
        _ => {break;}
    }
}
```

```
le let Some(value) = sms {
    if value > 0 {
        println!("algo");
        sms = Some(value -1);

    } else {
        println!("no hace algo");
        sms = None;
    }
}
```

Flujos de control

Ejercicios prácticos 9.

1. Ejercicio: Asigna un valor de Some(42) a una variable `numero` y usa `if let` para imprimir el valor si está presente.
2. Ejercicio: Crea una variable `resultado` con valor Ok("¡Éxito!") y usa `if let`,para imprimir el mensaje si es exitoso.
3. Ejercicio: Crea una lista de palabras ["hello", "world", "rust"] y usa `while let`,para imprimir cada palabra hasta que la lista esté vacía.
4. Ejercicio: Crea un vector con números del 1 al 5 y usa `while let` para sumar,todos los números del vector hasta que esté vacío.

“



Vector

Inicio

1. Han de guardar valores iguales mismo tipo.
 1. `let mut v: Vec<i32> = Vec::new();`
 2. `let mut v= vec![1,2,3];` esta forma es para hacer la vida mas fácil y es una macro.
2. Se libera la memoria al termina el scope
3. Para acceder se puede con posición (ojo si accedes a una que no exista) o con el método get

Vector

Referencias.

```
fn main() {  
    let mut v = vec![1, 2, 3];  
  
    let first = &v[0]; // Referencia a un elemento  
    println!("El primer elemento es: {}", first);  
  
    let second = &mut v[1]; // Referencia mutable  
    *second += 10; // Modificando el valor a través de la referencia mutable  
  
    println!("El vector modificado es: {:?}", v);  
}
```

Vector

Iteración y Modificación de Vectores

- Iterar sobre referencias inmutables
- Modificar elementos con referencias mutables

```
for i in &v2 {  
    println!("El valor de v2: {}", i);  
}  
  
for i in &mut v2 {  
    *i += 10;  
}  
  
for i in &v2 {  
    println!("El valor de v2 después: {}", i);  
}
```

Vector

Iteración y Modificación de Vectores

- Uso de enum para almacenar diferentes tipos

```
//esta es la forma de poder tener un vector con mas de un tipo de datos, ademas es una forma muy segura
enum Mensajes {
    TEXT0(String),
    ERROR(i32),
}

let mensajes: Vec<Mensajes> = vec![Mensajes::TEXT0("HOLA RUST".to_string()), Mensajes::ERROR(404)];
for m: &Mensajes in &mensajes{
    match m {
        Mensajes::TEXT0(texto: &String) => println!("valor: {}", texto),
        Mensajes::ERROR(codigo_error: &i32) => println!("valor: {}", codigo_error),
    }
    //println!("el valor de v despues {}", m)
}
```

HashSet

Conjunto de elementos únicos

- Colección que almacena elementos sin duplicados.
- Basado en una tabla hash para acceso rápido.
- Uso ideal para comprobaciones de pertenencia y eliminación de duplicados.

```
use std::collections::HashSet;

let mut frutas = HashSet::new();
frutas.insert("manzana");
frutas.insert("banana");
frutas.insert("naranja");
```

HashSet

Operaciones comunes con HashSet

- Agregar elementos: insert
- Eliminar elementos: remove
- Comprobar pertenencia: contains
- Iterar sobre elementos: iter

```
if frutas.contains("manzana") {  
    println!("La manzana está en el conjunto.");  
}  
  
frutas.remove("banana");
```

Hashmaps

¿Qué es un HashMap?

- Colección que almacena pares clave-valor.
- Basado en una tabla hash para acceso rápido.
- Uso ideal para asociaciones rápidas y búsqueda de claves.

```
use std::collections::HashMap;

let mut puntajes = HashMap::new();
puntajes.insert("Alice", 10);
puntajes.insert("Bob", 20);
```

Hashmaps

Operaciones con HashMap

- Agregar o actualizar elementos: insert
- Eliminar elementos: remove
- Acceder a valores: get
- Iterar sobre elementos: iter

```
if let Some(puntaje) = puntajes.get("Alice") {  
    println!("El puntaje de Alice es: {}", puntaje);  
}  
  
puntajes.remove("Bob");
```

“



Estructuras .10

Suma de Elementos en un Vector

1. Crea un programa en Rust que permita al usuario ingresar una lista de números enteros y luego calcule la suma de estos números.
 - Tips:
 - Usa el método `iter()` para iterar sobre el vector.
 - Usa `sum` para sumar los elementos del vector.
- ```
// para leer la linea
io::stdin().read_line(&mut input).expect("Error al leer linea");
// .split_whitespace() para separar por espacios
```

# Estructuras . 11

## Contar Elementos Únicos con HashSet

- Crea un programa que lea una serie de palabras ingresadas por el usuario y luego cuente cuántas palabras únicas hay.
- Tips:
  - HashSet automáticamente elimina duplicados.
  - Usa el método len para obtener el número de elementos en el HashSet.

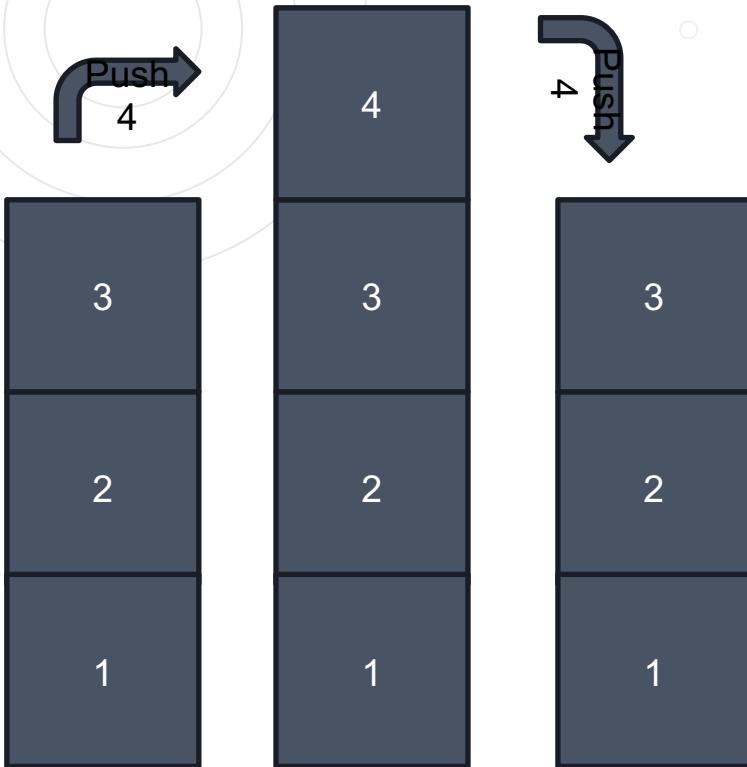
# Estructuras .12

## Buscar Valores en un HashMap

- Crea un programa que permita al usuario ingresar pares de claves y valores y luego buscar el valor asociado a una clave específica.
- Tips:
  - Usa `HashMap::insert` para insertar pares clave-valor.
  - Usa `HashMap::get` para buscar valores asociados a una clave.

# Stack

- Stack : Lo usa para almacenar información
- Al momento de hacer push se guarda arriba de la pila y es muy efectivo
- Ha de tener tamaño conocido : numero i32, f64 , bool, etc.. str



# Heap

- Heap : Es un poco mas desordenado y lo que hace es que busca memoria libre para guardar información y lo hace sin orden, es mas lento.
- Eso si podemos aumentar tamaño, String, vectores, etc..
- Todo aquello que puede ampliar información.





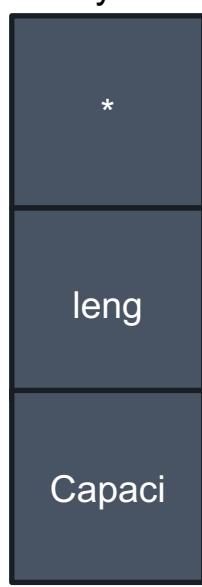
# Ownership Rules

- Cada Valor contiene un owner
  - Variable ejemplo
- Solo un owner por valor
- Cuando un owner sale del scope es eliminada
- VAMOS CON UN EJEMPLO !!!!

X



y

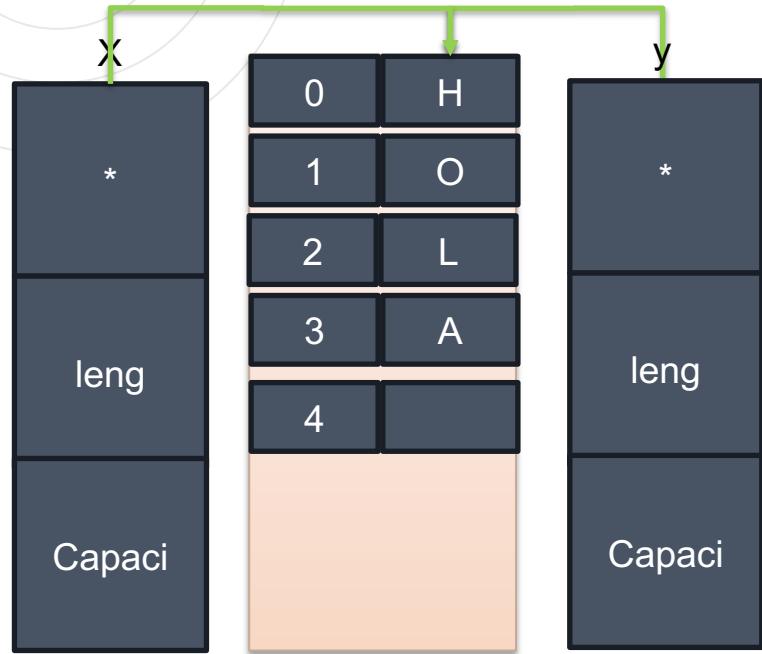


Esto invalida a x

rompe una de las reglas :

solo un dueño por valor

Al limpiar la memoria de x no hay nada luego y pero ya  
no hay nada y habría fallos de memoria



Por que no lo clona , por tema de coste  
Si tuviera un vector con 10 millones de registros  
Al clonar tendríamos 20 millones con información igual

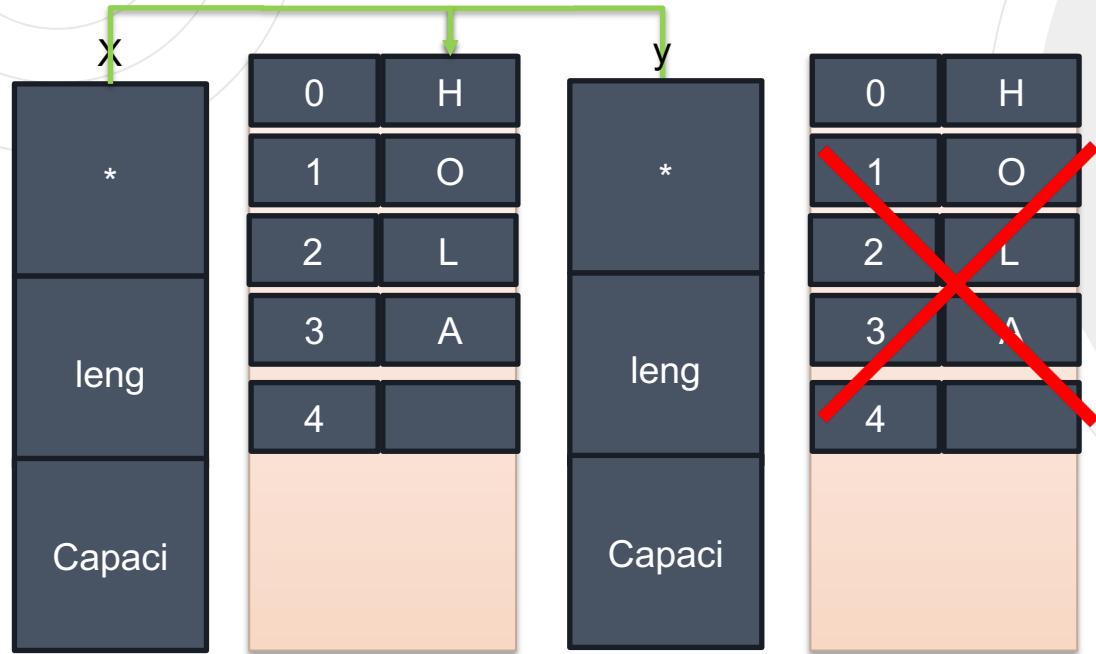


Esto invalida a x

rompe una de las reglas :

solo un dueño por valor

Al limpiar la memoria de x no hay nada luego y pero ya  
no hay nada y habría fallos de memoria



Por que no lo clona , por tema de coste

Si tuviera un vector con 10 millones de registros

Al clonar tendríamos 20 millones con información igual

# Lifetimes

¿Qué son los lifetimes?

Lifetimes son anotaciones que aseguran la validez de las referencias en Rust.

Garantizan que las referencias apunten a datos válidos durante un tiempo determinado.

Importancia en Rust

- Evitan problemas de seguridad como referencias nulas o inválidas.
- Ayudan al compilador a realizar comprobaciones estáticas en tiempo de compilación.

# Lifetimes

## Sintaxis Básica de Lifetimes

- Annotaciones de Lifetimes

- '*a*, '*b*, etc., son nombres de lifetimes.

- Se colocan antes de los tipos de referencia (&*'a T*).

- Reglas Generales

- Los lifetimes deben aparecer en la declaración de la función y en las referencias.

```
fn ejemplo<'a>(param: &'a i32) -> &'a i32 {
 // código
}
```

# Lifetimes

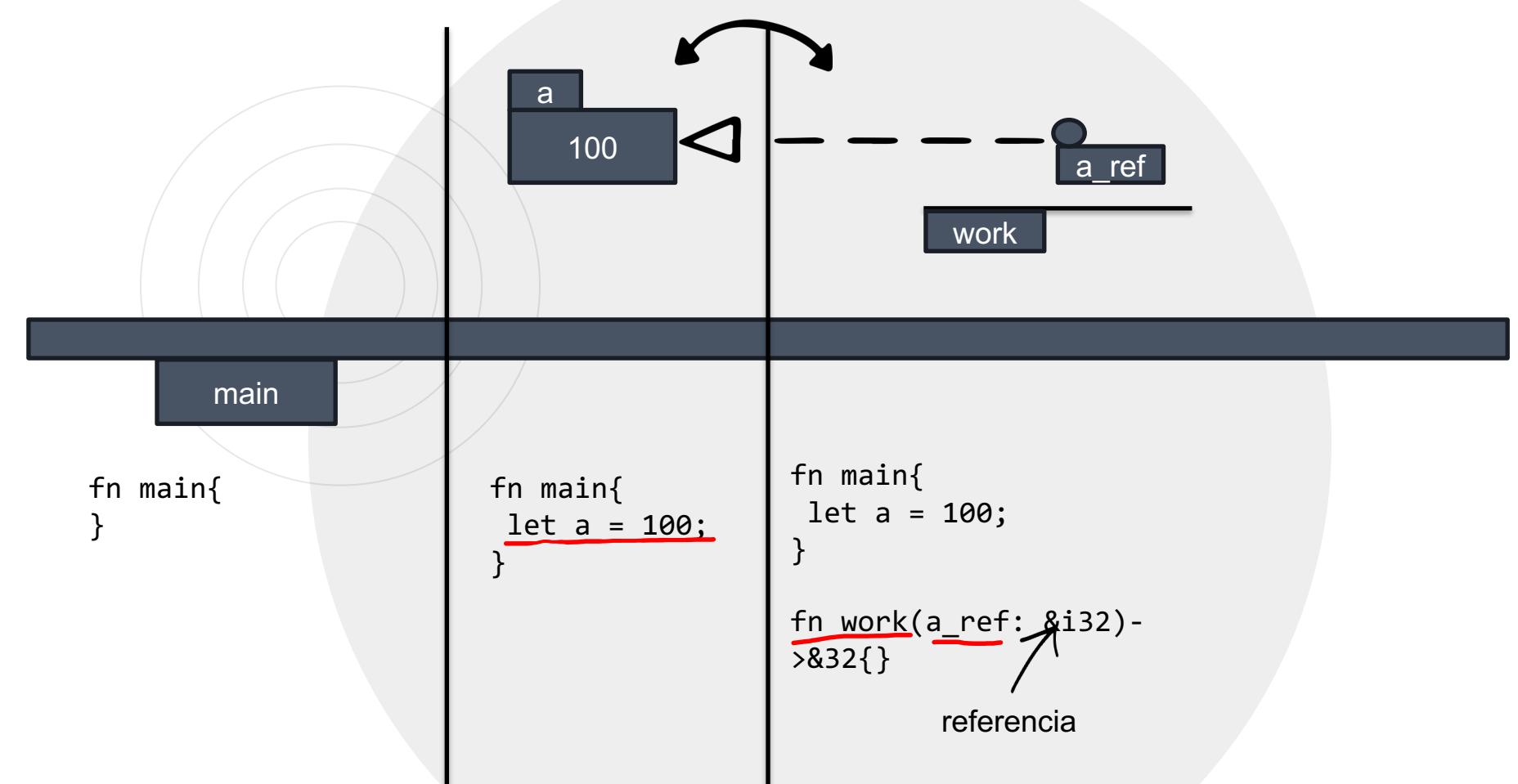
## Conclusiones

- Resumen de Lifetimes en Rust

- Lifetimes son esenciales para garantizar la seguridad y la integridad de las referencias en Rust.  
Permiten al compilador realizar verificaciones estáticas en tiempo de compilación.

- Reglas Generales

- Los lifetimes deben aparecer en la declaración de la función y en las referencias.



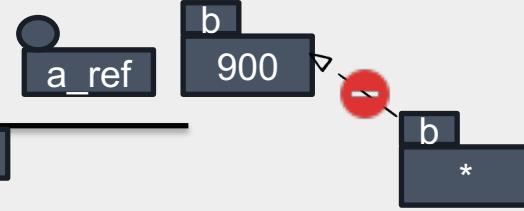


work

main

```
fn main{
 let a = 100;
}
```

```
fn work(a_ref: &i32)->&i32{
 let b = *a_ref + 900
}
```

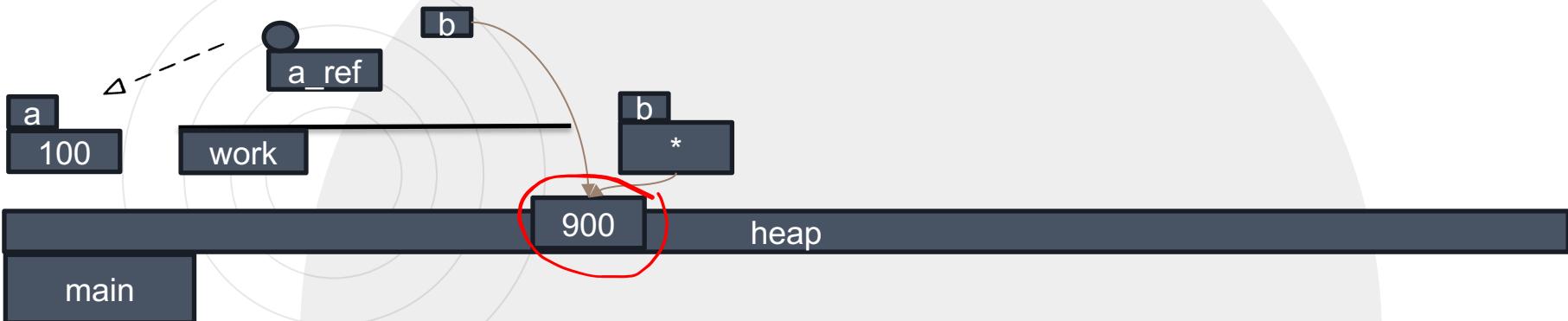


work

main

```
fn main{
 let a = 100;
 let b_ref = work(&a)
}
```

```
fn work(a_ref: &i32)->&i32{
 let b = *a_ref + 900
 &b
} error [E0515] :cannot
return reference to local
variable b
```

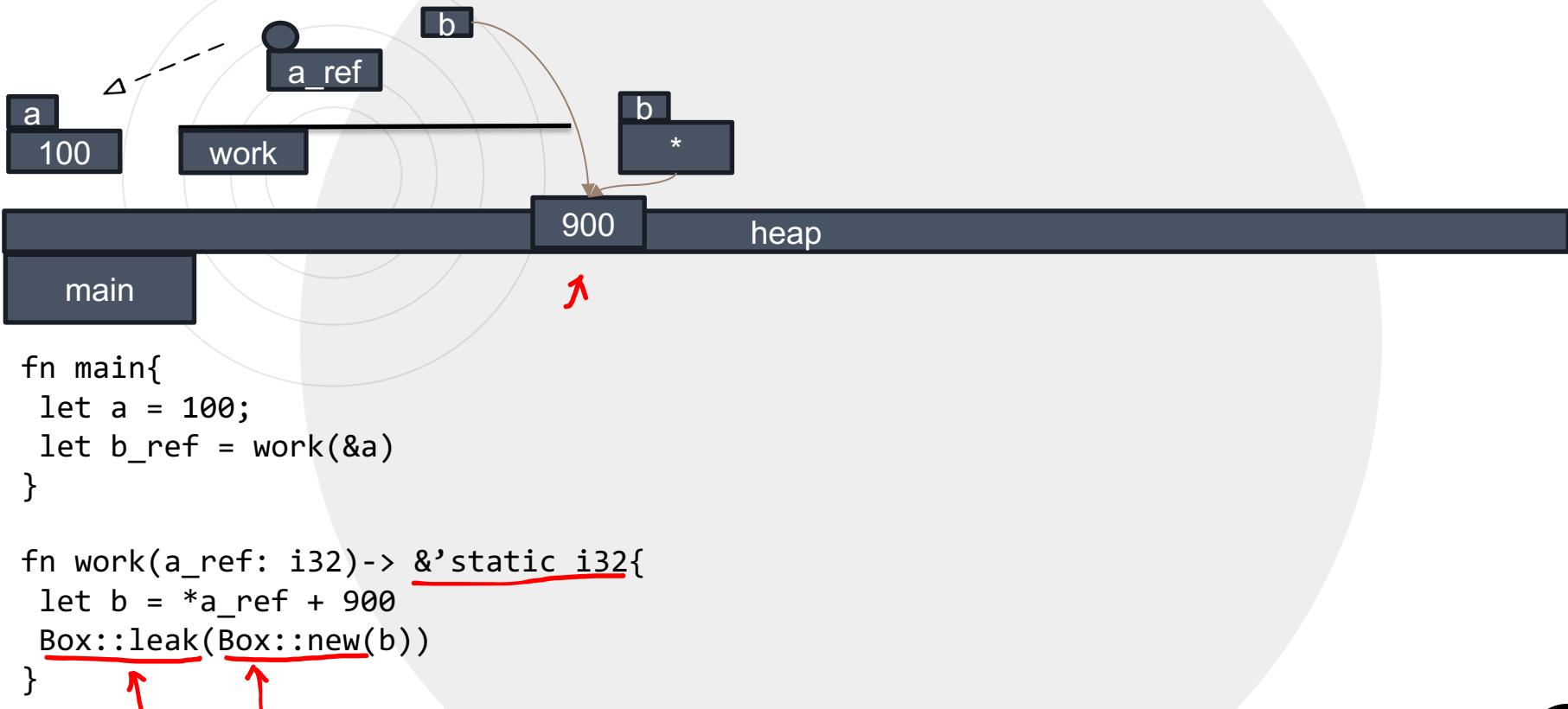


```
fn main{
 let a = 100;
 let b_ref = work(&a)
}
```

```
fn work(a_ref: i32) ->
 Box {
 let b = *a_ref + 900
 Box::new(b)
 }
```

```
fn main{
 let a = 100;
 let b_ref = work(&a)
}
```

```
fn work(a_ref: i32)c{
 let b = *a_ref + 900
 Box::new(b)
}
```





```
fn main{
 let a = 100;
 let b_ref = work(&a)
}
```

```
fn work(a_ref: &mut i32{
 *a_ref = *a_ref + 900
}
```

# Ejemplos

“



# Lifetimes

## Ejerciceos 13

- Función que Devuelve una Referencia
  - La función mayor toma dos referencias a i32 con el mismo lifetime 'a y devuelve una referencia al número mayor.
  - El uso de lifetimes asegura que las referencias sean válidas durante el tiempo que se necesiten.

# Lifetimes

## Ejerciceos 14

- Vamos a definir un struct que contiene una referencia a un str, y una función que devuelve una referencia a un campo del struct.
- El struct Persona tiene un campo nombre que es una referencia con un lifetime 'a.
- La función obtener\_nombre devuelve una referencia al campo nombre del struct, asegurando que la referencia sea válida mientras la instancia de Persona sea válida.

# Lifetimes

## Ejerciceos 15

- Vamos a escribir una función que toma un String y devuelve una referencia a su contenido. Este ejemplo ilustra cómo el manejo incorrecto del ownership puede causar problemas de compilación.
- La función `dame_ownership` toma una referencia a un String y devuelve una referencia al contenido del String.
- Aquí se muestra cómo manejar correctamente las referencias para evitar problemas de ownership.

# Smart Pointers

## Box<T>

- Punteros inteligentes en Rust son estructuras de datos que actúan como punteros, pero con funcionalidades adicionales.
- Propósito: Gestión segura y eficiente de recursos, automatizando la liberación de memoria.
- Tipos comunes:
  - Box<T>
  - Rc<T>
  - Arc<T>
  - RefCell<T>

# Smart Pointers

## ¿Qué es Box<T>?

- Box<T> es un puntero inteligente que asigna datos en el heap.
- **Uso principal:** Situaciones donde necesitamos un tamaño de datos fijo en tiempo de compilación pero queremos almacenar datos en el heap.

```
let b = Box::new(5);
```

# Smart Pointers

## Ventajas de Box<T>

- Asignación en el heap:
  - Útil para datos grandes o estructuras de tamaño desconocido en tiempo de compilación.
- Tamaño fijo en el stack:
  - El puntero Box<T> siempre tiene un tamaño conocido, independientemente del tamaño de los datos en el heap.
- Transfiere la propiedad:
  - El valor en un Box<T> puede ser transferido, garantizando la propiedad única y evitando problemas de doble liberación de memoria.

# Smart Pointers

## Box<T> en Recursión

- Box<T> es útil para estructuras de datos recursivas.

```
enum List {
 Cons(i32, Box<List>),
 Nil,
}

use List::{Cons, Nil};

let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
```

# Ejercicio 16

- Modificar el ejemplo para almacenar una estructura más compleja en Box.

## Explicación

- **Box:** Box<T> es un puntero inteligente que asigna memoria en el heap.
  - **new:** El método Box::new crea una nueva instancia de Box, almacenando el valor en el heap.
  - **Desreferenciación:** Se usa \* para desreferenciar Box y obtener el valor almacenado en el heap.
- **Tip**
    - **Memoria en el Heap:** Box se utiliza cuando se necesita almacenar datos en el heap en lugar de la pila. Es útil para grandes estructuras de datos o para valores cuyo tamaño no se conoce en tiempo de compilación.

# Smart Pointers

## Propiedades, Seguridad y Limitaciones

- Propiedad única:Box<T> garantiza que solo un propietario existe a la vez.
- Desempaque seguro:Cuando un Box<T> sale del alcance, su contenido se libera automáticamente.
- No Copy:Box<T> no se puede copiar, lo que evita duplicar referencias al mismo recurso.
- Sobrehead de heap:La asignación y desasignación de memoria en el heap es más costosa que en el stack.
- Propiedad única:No se pueden compartir datos mutables. Para compartir, considere Rc<T> o Arc<T>.

# Smart Pointers

## Rc (Reference Counting) en Rust

- Rc es un tipo de puntero inteligente en Rust que permite la **compartición de datos** entre múltiples partes del código sin tomar posesión única de ellos.
- Es útil cuando deseas compartir datos de solo lectura entre varias partes del programa sin necesidad de concurrencia.
- **Limitación:** No es seguro para el acceso concurrente desde múltiples hilos.

# Smart Pointers

## Arc (Atomic Reference Counting) en Rust

- Arc es una variante de Rc que es segura para el acceso concurrente entre múltiples hilos.
- Utiliza contadores de referencia atómicos para permitir la compartición de datos de forma segura en un entorno concurrente.
- Ideal para situaciones en las que los datos compartidos se leen desde varios hilos.

# Smart Pointers

## Mutex (Mutual Exclusion) en Rust

- Mutex proporciona **exclusión mutua** para proteger el acceso a datos compartidos en un entorno concurrente.
- Garantiza que solo un hilo puede acceder a los datos protegidos en un momento dado, evitando condiciones de carrera.
- Se utiliza en combinación con Arc para compartir datos seguros entre hilos.

# Smart Pointers

## Traí Dereferencia

- **Función:** Permite la desreferenciación (\*) de una estructura personalizada.
  - **Beneficio:** Permite que las estructuras personalizadas se comporten como referencias.
1. **MiCaja:** Estructura que envuelve un valor de tipo genérico T.
  2. **Método new:** Crea una nueva instancia de MiCaja.
  3. **Implementación de Dereferencia:** Permite usar \* para obtener una referencia al valor interno de MiCaja.
  4. **Uso en main:** Compara x y \*y con 5, mostrando que \*y se comporta como x.

# Smart Pointers

## Traí Dereferencia

- **Función:** Permite la desreferenciación (\*) de una estructura personalizada.
  - **Beneficio:** Permite que las estructuras personalizadas se comporten como referencias.
1. **MiCaja:** Estructura que envuelve un valor de tipo genérico T.
  2. **Método new:** Crea una nueva instancia de MiCaja.
  3. **Implementación de Dereferencia:** Permite usar \* para obtener una referencia al valor interno de MiCaja.
  4. **Uso en main:** Compara x y \*y con 5, mostrando que \*y se comporta como x.

# Smart Pointers

## Trai Deref Ejerciceos 17

- **Ejercicio 1:** Implementar Deref para una estructura CajaDoble<T> que contenga dos valores de tipo T.
- **Ejercicio 2:** Crear una estructura MiPuntero<T> que implemente Deref y DerefMut para permitir la mutabilidad.
- **Ejercicio 3:** Modificar el ejemplo para utilizar una estructura anidada CajaCaja<T> que contenga una MiCaja<T>

# Smart Pointers

## Traí Deref Ejerciceos

- Ejercicio 1: Implementar Deref para una estructura CajaDoble<T> que contenga dos valores de tipo T.
- Explicación
  - CajaDoble: Estructura que contiene dos valores de tipo genérico T.
  - Método new: Constructor para inicializar CajaDoble con dos valores.
  - Implementación de Deref: Permite usar \* para obtener una referencia al primer valor de CajaDoble.

# Smart Pointers

## Traí Deref Ejerciceos

- Ejercicio 2: Crear una estructura MiPuntero<T> que implemente Deref y DerefMut para permitir la mutabilidad.

### Explicación

1. **MiPuntero:** Estructura que envuelve un valor de tipo genérico T.
  2. **Método new:** Constructor para inicializar MiPuntero con un valor.
  3. **Implementación de Deref:** Permite usar \* para obtener una referencia al valor interno de MiPuntero.
  4. **Implementación de DerefMut:** Permite usar \* para obtener una referencia mutable al valor interno de MiPuntero.
- **Tip**
    - **DerefMut:** Similar a Deref, pero permite obtener una referencia mutable. Es útil cuando se necesita modificar el valor interno del puntero inteligente.

# Smart Pointers

## Trai Deref Ejerciceos

- Explicación

CajaCaja: Estructura que envuelve una instancia de MiCaja.

Método new: Constructor para inicializar CajaCaja con un valor.

Implementación de Deref: Propaga la desreferenciación a través de MiCaja, permitiendo usar \* para obtener el valor interno.

### Conceptos Clave

- Deref

- Propósito: Permite la desreferenciación (\*) de una estructura personalizada, haciendo que se comporte como una referencia.
- Uso: Implementado para devolver una referencia al valor interno.

- DerefMut

- Propósito: Similar a Deref, pero permite obtener una referencia mutable al valor interno.
- Uso: Implementado para devolver una referencia mutable, permitiendo modificar el valor interno.

# Smart Pointers

## ¿Qué es el Trait Drop?

- El trait Drop permite personalizar lo que sucede cuando un valor sale del alcance (scope).
- Es análogo a un destructor en otros lenguajes de programación.
- Se utiliza para liberar recursos, como memoria o archivos, de manera segura.

# Smart Pointers

## Implementaciòn Basica

```
struct MiRecurso;

impl Drop for MiRecurso {
 fn drop(&mut self) {
 println!("MiRecurso se está liberando");
 }
}

fn main() {
 let _recurso = MiRecurso;
 println!("Fin del scope principal");
}
```

# Smart Pointers

## Implementaciòn Basica

- Manejo de Memoria: Liberar memoria asignada manualmente.
- Archivos: Cerrar archivos abiertos.
- Conexiones: Terminar conexiones de red o bases de datos.

## Implementaciòn Basica

- No Panicar: Evitar que el mètodo drop cause un pánico (panic).
- Orden de Liberaciòn: Rust asegura que los valores se liberan en orden inverso a su creaciòn.
- Anidamiento: Los campos de una estructura se liberan despu s del contenedor.

# Smart Pointers

## Drop en Punteros Inteligentes

- Box: Libera la memoria en el heap cuando el Box sale del alcance.
- Rc: Libera la memoria cuando el contador de referencias llega a cero.
- Arc: Similar a Rc, pero seguro para hilos.

```
fn main() {
 let _box = Box::new(5);
 println!("Box creado");
 // `_box` se libera aquí, llamando a Drop para Box
}
```

# Smart Pointers

## Conclusion

- Trait Drop: Personaliza la lógica de liberación de recursos.
- Uso Común: Manejo de memoria, archivos y conexiones.
- Reglas: No causar pánicos, liberar en orden inverso, manejar anidamiento

# Smart Pointers

## Trait Drop Ejercicios 18

- Define una estructura Archivo.
- Implementa Drop para cerrar el archivo al salir del alcance.
- Tip:

```
let archivo = Archivo { nombre: String::from("data.txt") };
```

# Smart Pointers

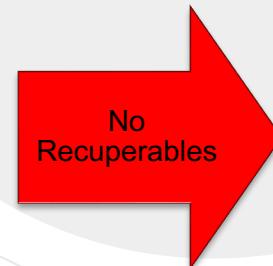
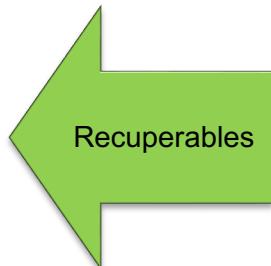
## Trait Drop Ejercicios 19

- Define una estructura Conexion.
- Implementa Drop para asegurar que la conexión se cierre correctamente.

# Manejo de Errores

## Panic y Result

- Recuperables como abrir un fichero
- No recuperables tratar de ir a una zona de memoria que no existe



# Manejo de Errores

## Panic y Result

- Recuperables: No tiene Excepción , lo que nos devuelve es un Result → Result<T,E>

- Los no recuperables nos dan un panic!

```
fn main(){
 panic!("explata")
}
```

# Manejo de Errores

## Panic y Result

```
use std::fs::File;
▶ Run | Debug
fn main(){
 // Tipo Result
 let file: Result<File, Error> = File::open(path: "algun/path");
 match file {
 Ok(file: File) => read_file(file),
 Err(error: Error) => println!("no puede abrir el archivo"),
 }
}

fn read_file(file: File){} unused variable: `file`
```



# Thanks!

**Any questions?**

You can find me at

@sowe

ravamo@gmail.com

# Duda