Rust 2025

clase 3

Temario

- Structs
- Enums
- Option
- Collections P1

Structs

Structs: ¿Qué son?

Es un tipo de dato personalizado que permite empaquetar y nombrar valores relacionados que forman un conjunto de datos. Son similares, en la programación orientada a objetos al conjunto de atributos que tiene una clase.

Structs: ¿Cómo se definen?

Se definen con la palabra clave struct de la siguiente manera:

```
struct NombreDelStruct{
    nombre_atributo_1: tipo1,
    nombre_atributo_2: tipo2,
    nombre_atributo_n: tipo_n
}
```

Structs: ¿Cómo se definen?

Su definición no necesariamente tiene que ser dentro de la función main

```
struct Persona{
    nombre: String,
    apellido: String,
    dni: i32
}
fn main() {
   let persona1= Persona{
         nombre:"Lionel".to_string(),
         apellido: "Messi".to_string(),
         dni:1,
   };
   println!("nombre: {} apellido:{} dni:{}", persona1.nombre, persona1.apellido, persona1.dni);
}
```

Structs: init shorthand

```
fn main() {
    let persona1= nueva_persona(
        "Lionel".to_string(),
        "Messi".to_string(),
        1
    );
    println!("nombre: {} apellido:{} dni:{}" , persona1.nombre, persona1.apellido, persona1.dni);
}
fn nueva_persona(nombre: String, apellido: String, dni: i32) -> Persona{
    Persona {
        apellido,
        dni,
        nombre,
    }
```

Structs: modificaciones

```
fn main() {
    let mut persona1= nueva_persona(
            "Lionel".to_string(),
            "Messi".to_string(),
            1
    );
    println!("nombre: {} apellido:{} dni:{}" , persona1.nombre, persona1.apellido, persona1.dni);
    persona1.dni = 99;
    println!("nombre: {} apellido:{} dni:{}" , persona1.nombre, persona1.apellido, persona1.dni);
}
```

Structs: creando instancias desde data de otra instancia

```
fn main() {
    let persona1= nueva_persona(
          "Lionel".to_string(),
          "Messi".to_string(),
          1
    );
    let persona2 = Persona{
          nombre:"Thiago".to_string(),
          ..persona1
    };
    println!("nombre: {} apellido:{} dni:{}", persona2.nombre, persona2.apellido,
persona2.dni);
 }
```

Structs: Tuple struct

```
struct Coordenada(f64, f64);

fn main() {
 let la_plata = Coordenada(-34.9213094, -57.9555699);
 println!("latitud: {} longirud:{}", la_plata.0, la_plata.1);
}
```

Structs: Implementando métodos (funciones asociadas)

```
struct Coordenada(f64, f64);
impl Coordenada {
  fn es_la_plata(self) -> bool{
      let (latitud,longitud)  = (-34.9213094, -57.9555699);
      if self.0==latitud && self.1 == longitud{
          return true;
      }
      false
  }
}
fn main() {
 let la_plata = Coordenada(-34.9213094, -57.9555699);
 println!("es la plata? {}", la_plata.es_la_plata());
}
```

Structs: Implementando métodos(func. asoc.), otro ej

```
struct Rectangulo {
    ancho: u32,
    altura: u32,
}
impl  Rectangulo{
    fn area(self) -> u32{
        self.ancho * self.altura
    }
}
fn main() {
  let rec1 = Rectangulo{ancho:3, altura:7};
  println!("el area del rectangulo es:{}", rec1.area());
}
```

Structs: Implementando métodos(func. asoc.), otro ej

```
fn main() {
    let rec1 = Rectangulo{ancho:3, altura:7};
    println!("rectangulo es:{}", rec1);
}
```

```
error[E0277]: `Rectangulo` doesn't implement `std::fmt::Display`
  --> src/main.rs:62:45
   |
62 |     println!("el area del rectangulo es:{}", rec1);
   |                                             ^^^^ `Rectangulo` cannot be formatted with the default form
atter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Rectangulo`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
   = 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)
```

Structs: Implementando métodos(func. asoc.), otro ej

```
fn main() {
    let rec1 = Rectangulo{ancho:3, altura:7};
    println!("el area del rectangulo es:{:?}", rec1);
}
```

```
error[E0277]: `Rectangulo` doesn't implement `Debug`
  --> src/main.rs:62:47
   |
62 |     println!("el area del rectangulo es:{:?}", rec1);
   |                                               ^^^^ `Rectangulo` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Rectangulo`
   = note: add `#[derive(Debug)]` to `Rectangulo` or manually `impl Debug for Rectangulo`
   = 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 annotating `Rectangulo` with `#[derive(Debug)]`
   |
14 | #[derive(Debug)]
```

Structs: Implementando métodos(func. asoc.), otro ej

```
#[derive(Debug)]
struct Rectangulo {
  ancho: u32,
  altura: u32,
}
fn main() {
 let rec1 = Rectangulo{ancho:3, altura:7};
 println!("rectangulo es:{:?}", rec1);
}
```

Structs: funciones asociadas

Todas las funciones definidas dentro de un bloque impl se denominan funciones asociadas. Están asociadas con el tipo que lleva el nombre de impl.

Podemos definir funciones asociadas que no tienen self como su primer parámetro (y por lo tanto no son métodos) porque no necesitan una instancia del tipo para trabajar.

Las funciones asociadas que no son métodos a menudo se usan como constructores por ej. que devolverán una nueva instancia de la estructura. Estos a menudo se suelen definir como new, pero new no es un nombre especial y no está integrado en el lenguaje.

Structs: funciones asociadas ejemplos

```
impl Rectangulo{
    fn area(&self) -> u32{
        self.ancho * self.altura
    }
    fn new(ancho: u32, altura: u32) -> Rectangulo{
        Rectangulo { ancho, altura}
    }
}
fn main() {
  let rec1 = Rectangulo::new(3,7);
  println!("rectangulo es:{:?}", rec1);
  println!("el area del rectangulo es:{}", rec1.area());
```

Enums

Enums: enumeration

Es un tipo de dato que permite definir distintas variaciones.

Para definirlo se utiliza la siguiente sintaxis:

```
enum NombreEnum{
        VARIACION1,
        VARIACION2,
        VARIACION3,
        ...
}
```

Enums: ejemplos

```
enum Rol{
    PADRE,
    HIJO,
}
struct Persona{
    nombre: String,
    apellido: String,
    dni: i32,
    rol:Rol,
}
fn main() {
   let per1 = Persona{nombre:"Lionel".to_string(), apellido:"Messi".to_string(), dni:1, rol: Rol::PADRE,};
   println!("el rol de:{} es:{:?}", per1.nombre, per1.rol);
}
```

Enums: ejemplos => con valores

```
enum Rol{
   PADRE(i32),
   HIJO(i32),
}
fn main() {
  let per1 = Persona{
   nombre:"Lionel".to_string(),
   apellido:"Messi".to_string(),
   dni:1,
   rol: Rol::PADRE(5),
  };
  match  per1.rol{
       Rol::PADRE(valor) => println!("{}", valor),
       _ => (),
  };
}
```

Enums: ejemplos => con Struct

```
struct StructPadre{}
struct StructHijo{}
impl StructPadre {
  fn hace_algo(self){
      println!("soy un padre que hace algo");
  }
}
impl StructHijo {
  fn hace_algo(self){
      println!("soy un hijo que hace algo");
  }
}
```

Enums: ejemplos => con Struct cont..

```
enum Rol{
    PADRE(StructPadre),
    HIJO(StructHijo),
}
impl Rol{
    fn hace_algo(self){
        match self {
            Rol::PADRE(instancia) => instancia.hace_algo(),
            Rol::HIJO(instancia) => instancia.hace_algo(),
        }
    }
}
```

Enums: ejemplos => con Struct cont..

```
fn main() {
    let per1 = Persona{
     nombre:"Lionel".to_string(),
     apellido:"Messi".to_string(),
     dni:1,
     rol: Rol::PADRE(StructPadre{}),
    };
    per1.rol.hace_algo();
 }
```

Option

Option: ¿Qué es?

Option es un enum que está disponible es la lib standard

Este enum tiene 2 posibles variantes que son Some() y None

Rust nos obliga a que en caso de que tengamos algún campo que no sepamos el valor, es decir vacío o nulo tenemos que manejar explícitamente y de manera obligatoria en código el caso. De esta forma se evitan los errores del tipo Null Pointer Exception de otros lenguajes.

Option: Ejemplo I

```
struct Persona{
    nombre: String,
    apellido: String,
    dni: Option<i32>,
    rol:Rol,
}
impl Persona {
    fn new(nombre:String, apellido:String, rol:Rol, dni:Option<i32>) -> Persona{
        Persona{
            nombre,
            apellido,
            dni,
            rol
        }
```

Option: Ejemplo I cont.

```rust
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = None;
    let per1 = Persona::new(nombre, apellido, rol, dni);
    println!("la persona:{} tiene el dni:{:?}" , per1.apellido, per1.dni);
}
```

Option: Ejemplo I con valor

```
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = Some(1);
    let per1 = Persona::new(nombre, apellido, rol, dni);
    match per1.dni {
     Some(valor) => println!("el dni de: {} es: {}", per1.apellido, valor),
     None => println!("{} no tiene nro de dni registrado", per1.apellido)
    }
 }
```

Option: Ejemplo II con otro struct

```
struct DNI{
    tipo: char,
    nro: u32,
}
struct Persona{
    nombre: String,
    apellido: String,
    dni: Option<DNI>,
    rol:Rol,
}
impl Persona {
    fn new(nombre:String, apellido:String, rol:Rol, dni:Option<DNI>) -> Persona{
        Persona{nombre, apellido, dni, rol}
    }
```

Option: Ejemplo II con otro struct cont.

```
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = Some(DNI{tipo:'A', nro:1});
    let per1 = Persona::new(nombre, apellido, rol, dni);
    match per1.dni {
     Some(valor) => println!("el dni de: {} es: {}", per1.apellido, valor.nro),
     None => println!("{} no tiene nro de dni registrado", per1.apellido)
    }
 }
```

Option: Ejemplo II con otro struct cont.

```
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = Some(DNI{tipo:'A', nro:1});
    let per1 = Persona::new(nombre, apellido, rol, dni);
    if per1.dni.is_none(){
        println!("{} no tiene nro de dni registrado", per1.apellido);
    }else{
          println!("el dni de: {} es: {:?}", per1.apellido, per1.dni.unwrap());
    }
}
```

Option: if let

```
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = Some(DNI{tipo:'A', nro:1});
    let per1 = Persona::new(nombre, apellido, rol, dni);
    if let Some(data) = per1.dni  {
        println!("el dni de: {} es: {}", per1.apellido, data.nro);
    }else{
        println!("{} no tiene nro de dni registrado", per1.apellido);
    }
}
```

Option: let else

```
fn main() {
    let nombre = "Lionel".to_string();
    let apellido = "Messi".to_string();
    let rol = Rol::PADRE(StructPadre {});
    let dni = Some(DNI{tipo:'A', nro:1});
    let per1 = Persona::new(nombre, apellido, rol, dni);
    let Some(data) = per1.dni else{
        panic!("{} no tiene dni", per1.apellido);
    };
}
```

Option: while let

```
fn main() {
    let mut cantidad = Some(5);
    loop{
        match cantidad {
            Some(valor) => {
                if valor > 0 {
                    println!("{}", valor);
                    cantidad= Some(valor -1);
                }else{
                    cantidad= None;
                }
            },
            None => {break;}
        }
    }
}
```

Option: while let cont.

```
fn main() {
    let mut cantidad = Some(5);
    while let Some(valor) = cantidad  {
        if valor > 0{
            println!("{valor}");
            cantidad = Some(valor -1);
        }else{
            cantidad = None;
        }
    }
}
```

Collections

- Primera Parte:
  - Sequences: Vec, VecDeque, LinkedList
  - Maps: HashMap, BTreeMap

Collections: ¿Qué son?

Son estructuras de datos que permiten almacenar y organizar datos de una manera flexible y dinámica.

Sequences: Vec

Vec: creación y agregando elementos

```rust
fn main() {
    //creacion
    let mut vector = Vec::new();
    //agregando elementos
    for i in 1..7{
        vector.push(i);
    }
    //recorriendolo
    for j in vector{
        println!("{j}");
    }
}
```

Vec: accediendo a elementos

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    // para acceder al primer elemento
    let primero = vector.first();
    if let Some(elemento) = primero {
        println!("El primer elemento es: {}", elemento);
    }
    println!("Tambien puedo acceder desde el indice: {}", vector[0]);
}
```

Vec: accediendo a elementos II

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    //para acceder al ultimo elemento
    let ultimo = vector.last();
    if let Some(elemento) = ultimo {
        println!("El ultimo elemento es: {}", elemento);
    }
    println!("Tambien puedo acceder desde el indice: {}", vector[vector.len()-1]);
}
```

Vec: agregando elementos II

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    //otra forma de agregar elementos
    let arreglo = [1,2,3];
    vector.extend(arreglo);
    println!("el ultimo elemento es:{}", vector[vector.len()-1]);
}
```

Vec: modificando elementos

```
fn main() {
    let mut vector = Vec::new();
    for i in 10..18{
        vector.push(i);
    }
    println!("{:?}", vector);
    //modificando elementos
    for i in 1..vector.len(){
        vector[i]+=4;
    }
    println!("{:?}", vector);
}
```

Vec: eliminando elementos

```
fn main() {
    let mut vector = Vec::new();
    for i in 1..7{
        vector.push(i);
    }
    println!("{:?}", vector);
    //eliminado un elemento de determinado indice
    vector.remove(1);
    println!("{:?}", vector);
}
```

Vec: simulando una pila

```
fn main() {
    let mut vector = Vec::new();
    for i in 1..7{
        vector.push(i);
    }
    //simulando una pila
    let elemento = vector.pop();
    if let Some(desapilado) = elemento {
        println!("desapilo el:{}", desapilado);
    }
    while let Some(desapilado) = vector.pop()  {
        println!("desapilo el:{}", desapilado)
    }
}
```

Vec: otras maneras de instanciarlos

```
fn main() {
    // otras formas de definirlos
    let  vector:Vec<i32> = vec![];
    let otro_vector = vec![1, 2, 3];
    let otro_mas_vector = vec![0;5];
    println!("{:?}", vector);
    println!("{:?}", otro_vector);
    println!("{:?}", otro_mas_vector);
}
//más data: https://doc.rust-lang.org/std/vec/struct.Vec.html
```

Sequences: VecDeque

VecDeque: agregando y sacando datos

Es una cola de doble atención, se puede agregar al final, al principio y sacar del final y del principio.

```
use std::collections::VecDeque;
fn main() {
    let mut buf = VecDeque::new();
    for i in 1..5{
        buf.push_back(i);
    }
     //[1,2,3,4,5]
    for i in 10..15{
        buf.push_front(i);
    }
    if let Some(numero) = buf.pop_front()  {
        println!("{numero}");
    }
    if let Some(numero) = buf.pop_back()  {
        println!("{numero}");
    }
}
```

VecDeque: accediendo a los datos

```
use std::collections::VecDeque;
fn main(){
    use std::collections::VecDeque;
    let mut deque: VecDeque<u32> = VecDeque::with_capacity(10);
    for i in 1..3{
        deque.push_front(i);
    }
    //puedo accerder por indice
    println!("{}", deque[0]);
    //puedo acceder por metodo get
    match deque.get(0) {
        Some(valor) => println!("{valor}"),
        _ =>()
    }//mejor con if let?
    println!("{:?}", deque);
}
```

VecDeque: modificando los datos

```rust
fn main(){
    use std::collections::VecDeque;
    let mut deque: VecDeque<u32> = VecDeque::with_capacity(10);
    for i in 1..3{
        deque.push_front(i);
    }
    //puedo hacerlo directamente por posición
    deque[0] = 22;
    // puedo hacerlo a traves del método get_mut
    if let Some(elem) = deque.get_mut(1) {
        *elem = 7;
    }
    println!("{:?}", deque);
}
//más data: https://doc.rust-lang.org/std/collections/struct.VecDeque.html
```

Sequences: LinkedList

LinkedList: creación y agregado de elementos

```
fn main(){
    use std::collections::LinkedList;
    let mut list1 = LinkedList::new();
    for i in 1..3{
        list1.push_back(i);
    }
    for i in 3..7{
        list1.push_front(i);
    }
}
```

LinkedList: operaciones más importantes

```
list1.back();// retorna un Option con el ultimo elemento si existe
list1.front();// retorna un Option con el primer elemento si existe
list1.back_mut();// retorna un Option con el ultimo elemento mutable si existe
list1.front_mut();// retorna un Option con el primer elemento mutable si existe
list1.clone();// clona la lista en una nueva lista
list1.contains(&4); // retorna un boolean si contiene o no el elemento
list1.is_empty();// retorna un boolean si está o no vacía
list1.len();// retorna la longitud de elementos de la lista
list1.pop_back();// retorna el ultimo elemento eliminandolo de la lista
list1.pop_front();// retorna el primer elemento eliminandolo de la lista
list1.clear();// limpia toda la lista y la deja vacía
//más data: https://doc.rust-lang.org/std/collections/struct.LinkedList.html
```

Maps: HashMap

HashMap: creación y agregado de elementos

```rust
fn main(){
    use std::collections::HashMap;
    let mut balances = HashMap::new();
    balances.insert(1, 10.0);
    balances.insert(2, 0.0);
    balances.insert(3, 150_000.0);
    balances.insert(4, 2_000.0);
    for (id, balance) in balances {
        println!("{id} tiene $ {balance}");
    }
}
```

HashMap: obtener y modificar un elemento

```
fn main(){
    let mut balances:HashMap<i32, f64> = HashMap::new();
    balances.insert(1, 10.0);
    balances.insert(2, 0.0);
    let id = 2;
    let balance:Option<&mut f64> = balances.get_mut(&id);
    match balance {
        Some(balance) => *balance = *balance + 12.0,
        _ => ()
    }
    if let  Some(balance)= balances.get(&id){
        println!("{balance}");
    }
}
```

HashMap: otra forma de construir

```
fn main(){
    let balances = HashMap::from([
        (1, 10.0),
        (2, 0.0),
    ]);
    // obtener solo las claves
    for id in balances.keys(){
        println!("{id}");
    }
    //obtener solo los valores
    for val in balances.values(){
        println!("{val}");
    }
}
```

HashMap: otros métodos importantes

```
balances.remove(&3);// elimina la clave-valor y retona un Option con el valor
balances.values_mut();// retorna los valores para poder modificarlos
balances.get_key_value(&1);// retorna un Option con el par clave-valor
balances.contains_key(&5);// retorna un bool
balances.entry(5).or_insert(0.0);// inserta la clave-valor solo si no existe
balances.len();
balances.is_empty();
balances.clear();
//más data: https://doc.rust-lang.org/std/collections/hash_map/struct.HashMap.html
```

Maps: BTreeMap

BTreeMap: ¿Qué son?

Los BTreeMap a diferencia de los HashMap tiene una pequeña mejora optimizada (Árbol Binario) en cuanto a búsqueda de la clave y su interfaz es igual a los HashMap. Es decir, podemos aplicar las mismas operaciones.

```
use std::collections::BTreeMap;
fn main(){
  let mut balances = BTreeMap::from([
        (1, 10.0),
        (2, 0.0),
  ]);
  //operaciones extra
  balances.pop_first();// remueve y retorna un Option con el elemento de clave mas pequeña
  balances.pop_last();// remueve y retorna un Option con el elemento de clave mas grande
  balances.first_key_value();
  balances.last_key_value();
}
//más data: https://doc.rust-lang.org/std/collections/struct.BTreeMap.html
```