Desarrollo de software para blockchain 2024

Clase 05

Rust

¿Qué es Rust?

Es un lenguaje de programación multiparadigma compilado de código abierto que se centra en la seguridad, la concurrencia y el rendimiento. Diseñado para ayudar a los desarrolladores a escribir código seguro y eficiente, ofrece características únicas que permiten un manejo de memoria seguro en tiempo de compilación sin la necesidad de un recolector de basura.

¿Qué es Rust? Algunas características

- **Sistema de tipos:** tiene un sistema de tipos estático y fuertemente tipado, lo que significa que el tipo de cada variable debe ser conocido en tiempo de compilación y no puede cambiar durante la ejecución del programa.

- **Seguridad de memoria:** garantiza la seguridad de memoria mediante un sistema de propiedad y préstamos, que ayuda a prevenir errores comunes como el uso después de liberar, doble liberación y corrupción de memoria.

- **Rendimiento:** compila el código directamente a código de máquina nativo y optimizado, lo que generalmente proporciona un alto rendimiento.

¿Qué es Rust? Algunas características cont..

- **Calidad y ayuda en los mensajes de errores:** Los mensajes de error que arroja el compilador de rust son muy descriptivos, detallados y en muchas oportunidades explican cómo subsanar el error.

- **Gestión de errores:** utiliza el tipo Result para manejar errores de manera explícita y segura. Esto puede resultar en un código más seguro y fácil de razonar.

- **Macros (metaprogramación):** admite macros para la generación de código en tiempo de compilación, lo que permite la creación de abstracciones personalizadas y la generación de código eficiente.

- **Empaquetado y administración de dependencias:** incluye cargo, una herramienta de construcción y administración de paquetes que facilita la gestión de dependencias y la construcción de proyectos.

Un poco de historia

- Se comenzó a trabajar en rust por el 2006 y la versión 1.0 fue liberada en el 2015 por eso es un lenguaje muy reciente.

- Inicialmente fue pensado para programación de sistemas (sistemas operativos, browsers, engine de videojuegos)

- Fue creado por un grupo de desarrolladores de mozilla

¿Para que se usa?

Aplicaciones de línea de comando

Sistemas operativos

Browser engine

Backend/Api

Escribir aplicaciones de bajo nivel para mejorar performance

Webassembly (wasm)

Sistemas embebidos, microcontroladores, iot

Blockchain

¿Qué es cargo?

Es el administrador de paquetes de Rust como npm o pip por ej. y también nos facilita la creación de proyectos.

¿Como usar cargo?

`cargo new nombre_del_proyecto`

Esto nos creará un proyecto con la siguiente estructura:
-un archivo llamado Cargo.toml
-un directorio src

otros comandos: `build, run, check`

crates: `https://crates.io`

Vs Code extensiones recomendadas

rust-analyzer

better-toml

crates

error lens

¿Cómo ..

-hacer comentarios

-definir una variable

-inmutabilidad, mutabilidad

-definir una constante

Tipos de datos

Tipado estático: el chequeo de los tipos de datos se hace en tiempo de compilación.

Se dividen en scalar types y compound types.

Tipos de datos (scalar types)

- Integer: con y sin signo
- Floating-Point(32 y 64 bits)
- Boolean
- Character

Tipos de datos (scalar types)

Integer:

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`

ej: `let numero: u32 = 42;`

Tipos de datos (scalar types)

Integer literals:

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

ej: `let num = 32_500;`

Tipos de datos (scalar types)

Operaciones:

```
fn main() {
    let suma = 7 + 3;
    let resta = 95.5 - 4.3;
    let producto = 4 * 30;
    let division = 56.7 / 32.2;
    let division2 = -5 / 3; // Resultado -1
    let resto = 43 % 5;
}
```

Tipos de datos (scalar types)

Boolean

```
fn main() {
    let t = true;
    let f: bool = false; // con explicito tipo
}
```

operaciones `&, &&, |, || , == , ^ , > , < , >= , <= , ! , !=`

Tipos de datos (scalar types)

Character

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // con tipo explicito
    let gatito = '😻';
}
```

El tipo de char de Rust tiene un tamaño de cuatro bytes y representa un valor Unicode, lo que significa que puede representar mucho más que solo ASCII. Las letras acentuadas; los caracteres chinos, japoneses, coreanos, emojis, etc.

Tipos de datos (compound types)

- String
- Tuple
- Array

Tipos de datos (compound types): strings

-str: Es un tipo de cadena de caracteres inmutable y de longitud fija.

-String: Es un tipo de cadena de caracteres que es mutable y de longitud variable.

Tipos de datos (compound types): strings

ejemplo:

```
fn main() {
    let str_fijo:&str = "Soy un string inmutable";
    let mut str_mutable:String = "Soy mutable".to_string();
    str_mutable += " concateno" ;
    println!("{}", str_mutable);
}
```

Tipos de datos (compound types): tuple

Es una forma de agrupar distintos valores y pueden tener distintos tipos.

```
fn main() {
    let mut tupla:(String, f32, u8) = ("hola".to_string(), 3.0, 3);
    tupla.0 = "cambio valor".to_string();
    println!("{}", tupla.0);
    println!("{:?}", tupla);
    let(hola, flotante, entero) = tupla;
    println!("{}", hola);
}
```

Tipos de datos (compound types): arrays

Son de tamaño fijo y tienen el mismo tipo de dato. ej:

```
fn main() {
    let arreglo = [1,2,3,5];
    println!("el tercer el elemento es: {}", arreglo[2]);
    let arreglo2 :[char ;2]= ['1', '2'];
    println!("el ultimo elemento es: {}", arreglo2.last().unwrap());
}
```

Uso de libs

En rust podremos hacer uso de libs o módulos para determinada acción y hacer reutilización de código, observemos el siguiente fragmento de código:

```
use std::io::stdin;
fn main() {
    println!("Ingrese su nombre: ");
    let mut nombre = String::new();
    stdin().read_line(&mut nombre).expect("Error al leer el nombre.");
    println!("Hola, {}!", nombre);
}
```

En la línea 1 importamos stdin de la lib standar de rust para poder leer desde teclado información.

stdin().read_line no devuelve un Result, ya lo veremos más adelante

el expect nos sirve para indicar en caso de que el Result tenga error

que mensaje arrojar al disparar un Panic!

Estructuras de control

Estructuras de control: if, if-else

```
if condicion_booleana {
        //hace algo porque condicion_booleana es true
}
```

```
if condicion_booleana {
        //hace algo porque la condicion_booleana es true
}else{
        // hace algo porque la condicion_booleana es false
}
```

Estructuras de control: if-else if

```
if condicion_booleana {
        //hace algo porque la condicion_booleana es true
    }else if otra_condicion{
        // hace algo porque otra_condicion es true
    }else{
        //hace algo porque otra_condicion y condicion_boleana son
false
    }
```

Estructuras de control: if con declaración let

```
let data = if condicion_booleana{ 20 } else { 0};
```

```
fn main() {
    let number: i32 = 10;
    let condicion_booleana: bool = number < 10;
    let data: i32 = if condicion_booleana{
        //pueden haber mas intrucciones
        println!("entro por aca!");
        number*number
        } else {
            let mut n: i32 = number;
            n *=2;
            n
        };
    println!("{}", data);
}
```

Estructuras de control: match

la forma de match es la siguiente:

```
match algun_valor {
        patron_que_cumple_algun_valor => //hace algo porque lo cumple,
        otro_patron => //hace algo porque lo cumple,
}
```

patrón puede ser: `literals, destructured arrays, enums, structs, tuples, variables, wildcards, placeholders`

Estructuras de control: match(con variables)

```
let number = 10;
match number {
    3 => println!("es tres o hace algo porque es 3"),
    7 => println!("es siete o hace algo porque es 7"),
    other => println!("hace algo porque porque no es 3 ni 7"),
}
```

Estructuras de control: match(variables-placeholder)

```
let number = 10;
match number {
    3 => println!("es tres o hace algo porque es 3"),
    7 => println!("es siete o hace algo porque es 7"),
    _ => println!("hace algo porque porque no es 3 ni 7"),
}
```

Estructuras de control: match

```
let number = 10;
match number {
    3 => println!("es tres o hace algo porque es 3"),
    7 => println!("es siete o hace algo porque es 7"),
    _ => (),
}
```

Estructuras de control: while

```
let mut number = 0;
while number < 10{
    println!("{number}");
    number +=2;
};
```

Estructuras de control: for

```
let arreglo = [1, 2, 3, 4, 5];
for elemento in arreglo {
    println!("el valor es: {elemento}");
}
```

Estructuras de control: for

```
let limite = 5;
for i in 1..limite+1 {
    println!("el valor es: {i}");
}
for i in (1..limite+1).rev() {
    println!("el valor es: {i}");
}
```

Funciones

Funciones

Como se observó estuvimos viendo una función: main. La definición de una función se realiza con la palabra reservada “fn” a continuación el nombre de la misma (snake case) y luego entre los paréntesis los argumentos. Entre las llaves el código propio del scope de la función.

```
fn mi_nueva_funcion(arg1: tipo, arg2: tipo, arg_n:tipo){

        //codigo propio del scope de la función

}
```

Funciones

```rust
fn mi_funcion( data:i32){
    println!("{data}");
}
```

```rust
fn mi_funcion( data:[i32; 7]){
    for i in data{
        println!("{i}");
    }
}
```

Funciones: retornado valores

```
fn mi_funcion( data:i32) -> i32{
    println!("{data}");
    return data
}
```

```
fn mi_funcion( data:i32) -> i32{
    println!("{data}");
    data
}
```

Ownership y Borrowing

Ownership y borrowing

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

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

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

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

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

Ownership y borrowing

Estas reglas son las siguientes:

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

Ownership y borrowing

Stack vs Heap:

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

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

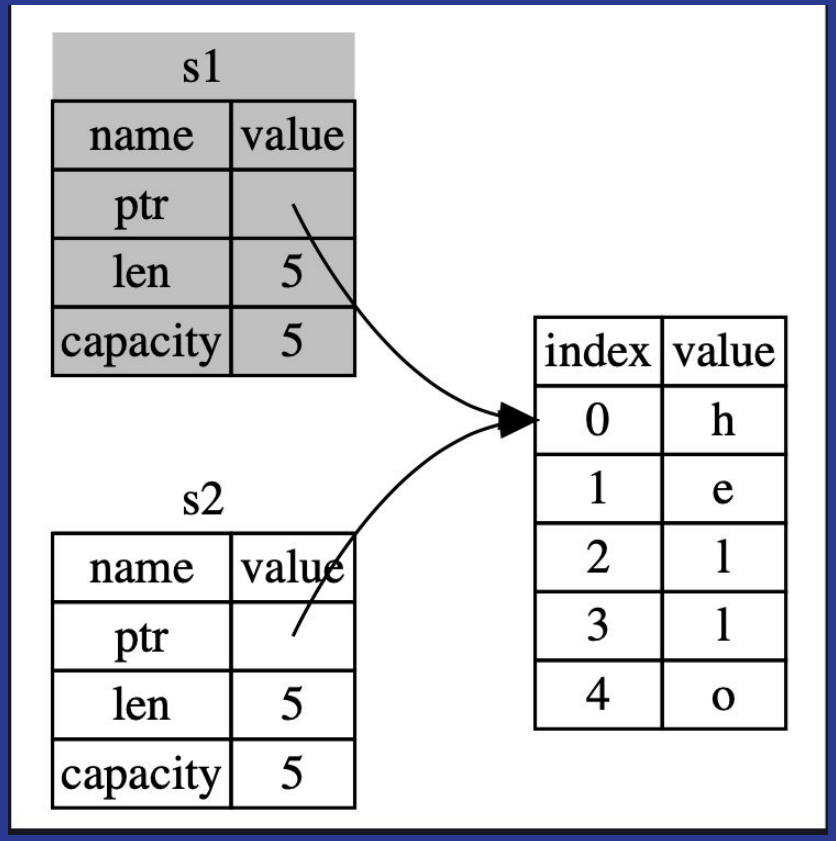
Ownership y borrowing

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

Ownership y borrowing

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

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


s1
name
value
ptr
len
5
capacity
5
s2
name
value
ptr
len
5
capacity
5
index
value
0
h
1
e
2
l
3
l
4
o

Ownership y borrowing

Que tipos implementan el trait Copy:

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

Ownership y borrowing: funciones

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

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

Ownership y borrowing: funciones

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

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

Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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

Lifetime

Lifetime

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

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

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

Lifetime: ejemplos

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

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

Lifetime: ejemplos

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

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

Lifetime: ejemplos

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

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

Lifetime: ejemplos

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

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

Lifetime: ejemplos

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

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

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

```
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, 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, 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, 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, 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.

```
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);
    }
}
```

Collections: ¿Qué son?

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

Secuences: 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: 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
```

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
```

Generics

Generics: ¿Para qué sirven?

El tipo generic se utiliza para generalizar implementaciones, permite mayor flexibilidad en el código

Generics: ejemplo I

```
#[derive(Debug)]
struct Punto{
    x:i32,
    y:i32,
}
fn main(){
    let p1 = Punto{x:1,y:2};
    println!("{:?}", p1);
}
```

Generics: ejemplo I

```
#[derive(Debug)]
struct Punto{
    x:i32,
    y:i32,
}
fn main(){
    let p1 = Punto{x:1,y:2};
    let p2 = Punto{x:10.5,y:2};
    println!("{:?} {:?}", p1, p2);
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:27:22
   |
27 |     let p2 = Punto{x:10.5,y:2};
   |                      ^^^^ expected `i32`, found floating-point number
```

Generics: ejemplo I

```
struct Punto<T>{
    x:T,
    y:T,
}
fn main(){
    let p1 = Punto{x:1,y:2};
    let p2 = Punto{x:10.5,y:2.0};
}
```

Generics: ejemplo I

```
struct Punto<T>{
    x:T,
    y:T,
}
fn main(){
    let p1 = Punto{x:1,y:2};
    let p2 = Punto{x:10.5,y:2};
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:10:29
   |
10 |     let p2 = Punto{x:10.5,y:2};
   |                             ^ expected floating-point number, found integer
```

Generics: ejemplo I

```
struct Punto<T, V>{
    x:T,
    y:V,
}
fn main(){
    let p1 = Punto{x:1,y:2};
    let p2 = Punto{x:10.5,y:2};
}
```

Generics: ejemplo I

```
#[derive(Debug)]
struct Punto<T>{
  x:T,
  y:T,
}
fn main(){
  let p1 = Punto{x:1,y:2};
  let p2 = Punto{x:10,y:2};
  println!("{:?}", p1);
  println!("{:?}", p2);
}
```

Generics: ejemplo de caja

```
struct Caja {
    dato:i32,
    estado:bool,
}
impl Caja {
    fn new(dato:i32) -> Caja{
        Caja { dato, estado:false}
    }
    fn abrir(&mut self)->i32{
        self.estado = true;
        self.dato
    }
    fn cerrar(&mut self){
        self.estado = false;
    }
}
```

Generics: otro ejemplo

```
fn main(){
    let mut caja = Caja::new(9);
    caja.abrir();
    caja.cerrar();
}
```

Generics: otro ejemplo

```
fn main(){
    let mut listado_de_compras = Vec::new();
    listado_de_compras.push((1, "Jabon de manos"));
    listado_de_compras.push((2, "Detergente"));
    let mut caja = Caja::new(listado_de_compras);
    caja.abrir();
    caja.cerrar();
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:24:30
   |
24 |     let mut caja = Caja::new(listado_de_compras);
   |                    ---------- ^^^^^^^^^^^^^^^^^^^ expected `i32`, found struct `Vec`
   |                    |
   |                    arguments to this function are incorrect
   |
   = note: expected type `i32`
              found struct `Vec<({integer}, &str)>`
```

```rust
struct Caja <T>{
    dato:T,
    estado:bool,
}
impl<T> Caja<T> {
    fn new(dato:T) -> Caja<T>{
        Caja { dato, estado:false}
    }
    fn abrir(&mut self)->&T{
        self.estado = true;
        &self.dato
    }
    fn cerrar(&mut self){
        self.estado = false;
    }
}
```

Traits

Trait ¿Qué es?

- Es una funcionalidad particular que tiene un tipo y puede compartir con otros tipos.
- Podemos usar traits para definir comportamiento de manera abstracta.
- Se pueden usar traits como límites en tipos de datos genéricos para determinada funcionalidad que el tipo genérico debe cumplir.
- Son similares a las interfaces llamadas en otros lenguajes pero con algunas diferencias.

Trait: ejemplo I

```
pub trait MulI32 {
    fn mul(&self, other:i32) -> f64;// abstracto
    fn hace_algo_concreto(&self){// por defecto
        println!("hace_algo_concreto");
    }
}
```

Trait : ejemplo I

```
impl MulI32 for f64{
    fn mul(&self, other:i32) -> f64{
        self * other as f64
    }
}
fn main(){
    let v1 = 2.8;
    let v2 = 4;
    let r = v1.mul(v2);
    println!("{}", r);
}
```

Trait : ejemplo II

```
struct Perro{}
struct Gato{}
fn main(){
    let gato = Gato{};
    let perro = Perro{};
    println!("{} {}", gato.hablar(), perro.hablar());
}
```

Trait : ejemplo II

```
pub trait Animal{
    fn hablar(&self) -> String;
}
impl Animal for Perro{
    fn hablar(&self) -> String{
        "Guau!".to_string()
    }
}
impl Animal for Gato{
    fn hablar(&self) -> String{
        "Miau!".to_string()
    }
}
```

Trait : limitando un generic

```
pub fn imprimir_hablar<T: Animal>(animal: &T) {
    println!("Hablo! {}", animal.hablar());
}
fn main(){
    let gato = Gato{};
    let perro = Perro{};
    imprimir_hablar(&gato);
    imprimir_hablar(&perro);
}
```

Trait : como parámetro

```
pub fn imprimir_hablar(animal1: &impl Animal, animal2: &impl Animal) {
    println!("Hablando! {} {}", animal1.hablar(), animal2.hablar());
}
fn main(){
    let gato = Gato{};
    let perro = Perro{};
    imprimir_hablar(&gato, &perro);
}
```

Trait : como parámetro

```
pub fn imprimir_hablar(animal: &impl Animal) {
    println!("Hablo! {}", animal.hablar());
}
fn main(){
    let gato = Gato{};
    let perro = Perro{};
    imprimir_hablar(&gato);
    imprimir_hablar(&perro);
}
```

Trait : múltiples

```
pub fn imprimir_hablar(animal: &(impl Animal + OtroTrait)) {
    println!("Hablo! {}", animal.hablar());
}
fn main(){
    let gato = Gato{};
    imprimir_hablar(&gato);
}
```

Trait : múltiples con where

```rust
pub fn imprimir_hablar<T>(animal: &T)
where
    T: Animal + OtroTrait
{
    println!("Hablo! {}", animal.hablar());
}
fn main(){
    let gato = Gato{};
    imprimir_hablar(&gato);
}
```

Manejo de errores

Manejo de errores

Rust agrupa los errores en recuperables e irrecuperables.
Un error recuperable es por ejemplo un archivo no encontrado, donde tan solo se informará el error pero la ejecución del programa continuará.
Los errores irrecuperables en cambio son siempre señales de bugs en nuestro código, como por ejemplo acceder a una posición inválida de un arreglo.
En la mayoría de los lenguajes no hay distinción entre estos 2 tipos de errores y suelen manejarse con excepciones.
Rust no tiene ni maneja excepciones, en su lugar tiene el tipo Result<T, E> para errores recuperables y la macro panic! para errores irrecuperables.

Manejo de errores: panic!

```rust
fn main() {
    let v = vec![1, 2, 3];
    v[v.len()];
}
```

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 3', src/main.rs:186:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Manejo de errores: panic!

export RUST_BACKTRACE=1 cargo run

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 3', src/main.rs:186:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/std/src/panicking.rs:579:5
   1: core::panicking::panic_fmt
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/panicking.rs:64:14
   2: core::panicking::panic_bounds_check
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/panicking.rs:159:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/slice/index.rs:260:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/slice/index.rs:18:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/alloc/src/vec/mod.rs:2703:9
   6: cripto::main
             at ./src/main.rs:186:5
   7: core::ops::function::FnOnce::call_once
             at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Manejo de errores: call panic!

```
fn verify(data: Vec<Data>){
    if data.is_empty(){
        panic!("No hay data para procesar y es obligatorio");
    }
    //hace mas cosas
}
```

Manejo de errores: Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Manejo de errores: Result

```
fn main() {
    use std::io::stdin;
    let mut buf = String::new();
    let result = stdin().read_line(&mut buf);
    match result {
        Ok(i) => procesar_entrada(i),
        Err(e) => procesar_error(e)
    }
}
```

Manejo de errores: Result

```rust
fn main() {
    let a = "123".to_string();
    let result = a.parse::<i32>();
    match result {
        Ok(data) => println!("el parseo fue correcto: {}", data),
        Err(e) => println!("String inválido para parsear a i32: {}", e)
    }
}
```

Alcance y visibilidad

Para definir a un elemento(`fn, struct, enum, mod`) como público se utiliza la palabra clave `pub`

delante de su definición, y esto indica que puede accedido desde fuera de donde fue declarado. En cambio si no se especifica con `pub` el compilador de rust hará que sea privado y solo puede accederse en donde fue definido.

Alcance y visibilidad

```
//ejemplo.rs
mod crate_helper_module {
    // esta función solo puede ser accedida en este rs
    pub fn crate_helper() {}

    //esta funcion solo puede ser accedida desde el scope de crate_helper_module
    fn implementation_detail() {}
}
// es es una funcion publica que puede ser accedida desde cualquier lugar, incluso
externamente
pub fn public_api() {}
```

Alcance y visibilidad

```rust
//ejemplo.rs
// Igual a public_api
pub mod submodule {
    use crate::ejemplo::crate_helper_module;
    pub fn my_method() {
        crate_helper_module::crate_helper();
    }
    // solo puede accederse en el scope de submodule
    fn my_implementation() {}
    #[cfg(test)]
    mod test {
        #[test]
        fn test_my_implementation() {
            // con super accedo jerarquicamente un nivel arriba de definiciones del módulo
            super::my_implementation();
        }
    }
}
```

Tests

Unit testing: en rust

```
#[test]
assert!(expresion);
assert_eq!(v1, v2);
assert_ne!(v1, v2);
```

Unit testing: en rust

```
comandos:
    cargo test
    cargo test nombre_del_test
    cargo test nombre_con_el_que_empieza

#[ignore]
#[should_be_panic(expected="mensaje del panic")]
```

Unit testing: coverage

Es una métrica utilizada en el contexto de pruebas de software que indica el porcentaje de código fuente que ha sido ejecutado durante la ejecución de las pruebas. Se utiliza para evaluar la efectividad de las pruebas en términos de qué tan bien cubren el código y qué áreas del código no están siendo probadas.

La cobertura de código tiene como objetivo identificar las áreas del código que no han sido probadas y que podrían contener errores o comportamientos inesperados. Una alta cobertura de código no garantiza la ausencia de errores, pero proporciona una mayor confianza en la calidad del software, ya que implica que se han realizado esfuerzos para probar exhaustivamente el código.

Unit testing: ventajas de coverage

- ★ **Identificación de código no probado:** La cobertura de código permite identificar las partes del código que no han sido ejecutadas durante las pruebas, lo que indica áreas de riesgo potencial donde los errores podrían estar presentes.
- ★ **Guía para la creación de pruebas:** La cobertura de código puede ayudar a guiar la creación de pruebas adicionales al revelar las áreas que requieren una mayor cobertura. Esto asegura que las pruebas se enfoquen en las partes críticas del código.
- ★ **Medición de la calidad de las pruebas:** La cobertura de código se utiliza como una métrica para evaluar la calidad de las pruebas. Una alta cobertura indica que se han realizado esfuerzos para probar exhaustivamente el código y puede indicar una mayor confiabilidad del software.

Unit testing: coverage en rust

tarpaulin: https://crates.io/crates/cargo-tarpaulin

comandos:

```
cargo tarpaulin --target-dir src/coverage --skip-clean --exclude-files=target/debug/*
cargo tarpaulin --target-dir src/coverage --skip-clean --exclude-files=target/debug/*
--out html
```