



Rust 2025



clase 2



Temario

- Estructuras de control
- Funciones
- Testing
- Borrowing
- Ownership
- Lifetime



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_booleana 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 instrucciones  
        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: loop

```
fn main() {  
    let mut number = 10;  
    loop{  
        number+=1;  
        if number == 30{  
            break;  
        }  
    }  
    println!("{}", number);  
}
```

Estructuras de control: loop

```
fn main() {  
    let mut number = 10;  
    let termina = loop{  
        number+=1;  
        if number == 30{  
            break true  
        }  
    };  
    println!("{}", number, termina);  
}
```

Estructuras de control: loop con tag

```
let mut count = 0;
'counting_up: loop {
    let mut remaining = 10;
    loop {
        if remaining == 9 {
            break;
        }
        if count == 2 {
            break 'counting_up;
        }
        remaining -= 1;
    }
    count += 1;
}
println!("End count = {count}");
```

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

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

```
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: aquí 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

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

```
error[E0382]: borrow of moved value: `s1`  
--> src/main.rs:14:20
```

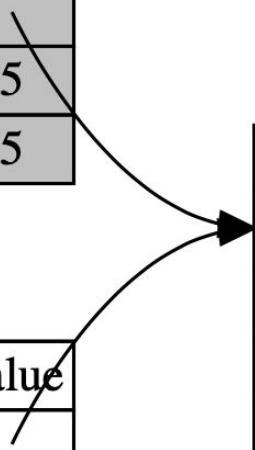
```
12 |     let s1= String::from("data ");  
    |           -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
13 |     let s2 = s1;  
    |           -- value moved here  
14 |     println!("{}", s1);  
    |                   ^^ value borrowed here after move
```

Ownership y borrowing

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

```
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 isn't necessary

--> src/main.rs:17:22

```
17 | fn mi_funcion(data: String){  
    |     ----- ^^^^^^ this parameter takes ownership of the value  
    |     |  
    |     in this function
```

= note: this error originates in the macro ``$crate::format_args_nl`` which comes from the expansion of the macro ``println`` (in Nightly builds, run with `-Z macro-backtrace` for more info)

help: consider cloning the value if the performance cost is acceptable

```
13 |     mi_funcion(dato1.clone());  
    |                   ++++++
```


Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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

Ownership y borrowing: funciones

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