

Estructuras de control

Toma de decisiones

- if
- if ... else
- else ... if y if anidado
- let ... if
- match

Bucles

Tipos de bucles

- Finitos
- Infinitos

- for
- loop
- while
- Break
- Continue

Estructuras de control

- Permiten controlar el flujo de ejecución del código.
- Son y/o contienen declaraciones y/o expresiones.
 - **Declaraciones:** Son instrucciones que *realizan alguna acción*, no retornan un valor y terminan en `;`, ejemplo: declarar una variable y su tipo, declarar una función.
 - **Expresiones:** Son instrucciones que *evalúan un valor resultante*, retornan un valor, si terminan en `;` se convierte en una declaración de expresión y no retornan un valor, ejemplo: números, caracteres, estructuras de control o llamadas a una función.
- Toda variable declara dentro del ámbito `{ ... }` de la estructura de control sólo es accesible dentro del ámbito de esta.

Toma de decisiones

- Se especifican condiciones a ser evaluadas, junto con un conjunto de declaraciones que se ejecutarán si la condición es verdadera o si es falsa.
- Permite bifurcar el código a partir de condiciones.
- **La condición debe ser de tipo booleano, no evalúa números.**

if

```
let numero=7;
if numero>0{
    println!("El número {} es positivo",numero);
}
```

- La expresión `if` consiste en la evaluación de una expresión booleana, si la expresión es verdadera se ejecuta un conjunto de declaraciones añadidos a la condición verdadera, si la expresión es falsa se salta el conjunto de declaraciones añadidos a la condición verdadera y ejecuta el código después del final del bloque `if`.

if ... else

```
let numero=6;
if numero%2==0{
    println!("El número {} es par",numero);
}else{
    println!("El número {} es impar",numero);
}
```

- Una expresión `if` puede o no ser seguida por una expresión `else`, es opcional.
- La expresión `else` ejecuta un bloque de código alternativo cuando la expresión booleana es falsa.

else ... if y if anidado

```
let numero=2;
if numero>0{
    println!("El número {} es positivo", numero);
    if numero%2==0 {
        println!("El número {} es par", numero);
    }else{
        println!("El número {} es impar",numero);
    }
}else if numero<0{
    println!("El número {} es negativo", numero);
    if numero%2==0 {
        println!("El número {} es par", numero);
    }else{
        println!("El número {} es impar",numero);
    }
}else{
    println!("El número es cero")
}
```

- Puedes usar la expresión `if` o `else if` dentro de otra expresión `if` o `else if`.
- Puntos a tener en cuenta:
 - Un `if` puede tener varios `else..if` y deben venir antes que el `else` si existiera.
 - Se evalúa secuencial-mente cada expresión `if`, si alguna expresión tiene éxito, todos los demás son descartados, es decir ninguno de los otros sera probado.

let ... if

```
let numero=42;
let resultado=if numero%7==0 {
    "es multiplo de 7"
}else{
    "no es multiplo de 7"
};
println!("El numero {} {}",numero,resultado);
```

- La expresión `if` es una expresión y podemos utilizarla como valor en una declaración `let`, para esto tenemos que hacer que el bloque de código de condición verdadera o falsa

termine en una expresión del mismo tipo de datos (Rust verifica en tiempo de compilación si es valido o no).

- La declaración `let` con la expresión `if` deben finalizar en `;`.

match

```
let codigo_aeropuerto="TPP";
let ciudad=match codigo_aeropuerto {
    "AQP"=>{
        println!("Se encontró una coincidencia para
{}",<div>codigo_aeropuerto);
        "Arequipa"
    },
    "CUZ"=>{
        println!("Se encontró una coincidencia para
{}",<div>codigo_aeropuerto);
        "Cusco"
    },
    "TPP"=>{
        println!("Se encontró una coincidencia para
{}",<div>codigo_aeropuerto);
        "Tarapoto"
    },
    "TBP"=>{
        println!("Se encontró una coincidencia para
{}",<div>codigo_aeropuerto);
        "Tumbes"
    },
    _=> "Desconocido"
};
if ciudad == "Desconocido"{
    println!("No se encontró ninguna coincidencia para
{}",<div>codigo_aeropuerto);
}else{
    println!("La ciudad es {}",<div>ciudad);
}
```

- Es similar a `switch` en el lenguaje C.
- La expresión `match` permite:
 - Comparar una variable con una lista de valores (arms).
 - Si el valor de la variable coincide con algún arm (brazo), se ejecuta un bloque o expresión de código añadido a la condición.
 - Ejecutar secuencial-mente cada arm, una vez que coincide y ejecuta un arm, no compara los siguientes arms.
- La expresión `match` es una expresión y podemos utilizarla como valor en una declaración `let`, para esto tenemos que cada bloque de código añadido a una condición termine en una expresión del mismo tipo de datos (Rust verifica en tiempo de compilación si es valido o no).
- La declaración `let` con la expresión `match` deben finalizar en `;`.
- El caso predeterminado se establece con guion bajo `_`, el cual coincidirá con todos los casos posibles que no se hayan especificado antes.

Bucles

- Son útiles cuando se requiere ejecutar un bloque de código o un conjunto de declaraciones repetidamente a partir de condiciones establecidas.
- Si la condición evaluada es verdadera, ejecuta secuencial-mente el código que se encuentra dentro del ámbito del bucle, luego vuelve a evaluar la condición y si es verdadera vuelve a ejecutar el código anterior desde el inicio, esto continua hasta que la condición evaluada sea falsa.
- Un bucle cuyo número de iteraciones está determinado, se le denomina bucle definido.

Tipos de bucles

Finitos

- Un bucle ejecuta un bloque de código durante un número específico de veces.
- Se puede utilizar para iterar sobre un conjunto fijo de valores, como un array.

Infinitos

- Un bucle ejecuta el bloque de código durante un número ilimitado o indefinido de veces.

for

```
let limite=7;
let mut factorial=1;
for contador in 1..limite{
    factorial*=contador;
    println!("numero: {} factorial: {}",contador, factorial);
}
for contador in (1..limite).rev(){
    println!("numero: {} factorial: {}",contador, factorial);
    factorial/=contador;
}
```

- Se recomienda su uso para el manejo de índices, debido a que ofrece mayor seguridad que el bucle `while`.
- Permite iterar entre números con range `..` y se puede revertir la iteración con el método `rev()`.
- Es el tipo de bucle más utilizado en el lenguaje Rust.

loop

```
let mut cantidad=0;
println!("se registraron un total de {} entradas", loop{
    let mut entrada=String::new();
    std::io::stdin().read_line(&mut entrada).expect("fallo al leer la
linea");
    if entrada.trim()=="exit" {
        break cantidad;
    }
    println!("valor recibido:{}",entrada);
    cantidad+=1;
});
```

- Es similar a un bucle infinito `while(true)`.
- Ejecuta un bloque de código para siempre o hasta que se lo indique con la expresión `break`.
- La combinación de `let ... break`, permite retornar la expresión donde se rompió el bucle, también es el único bucle donde se permite su uso.

while

```
println!("cantidad de numeros a sumar:");
let mut limite=String::new();
std::io::stdin().read_line(&mut limite).expect("fallo al leer la linea");
let limite=limite.trim().parse::<i32>().unwrap();
let mut acumulador=0;
let mut contador=0;
while contador<limite{
    let mut entrada=String::new();
    std::io::stdin().read_line(&mut entrada).expect("fallo al leer la
linea");
    let numero=entrada.trim().parse::<i32>().unwrap();
    acumulador+=numero;
    contador+=1;
}
println!("el resultado de la suma es: {}",acumulador);
```

- Permite evaluar una condición dentro de un bucle.
- Elimina gran cantidad de anidación si ha utilizado el patrón `loop.. if... else y break`.
- **Es propenso a errores**, porque podría entrar en pánico si el índice administrado sobrepasa la longitud de su estructura de datos.
- **Es lento**, porque el compilador agrega código de tiempo de ejecución para realizar la verificación condicional en cada elemento en cada iteración a través del bucle.

Break

- La expresión `break` permite interrumpir la ejecución del bucle y con esto quitar el control a la estructura.

Continue

```
for numero in 1..10 {
    if numero%2 ==0{
        continue;
    }
    println!("el numero {} es impar",numero);
}
```

- La expresión `continue` omite las declaraciones posteriores y lleva el control al principio del bucle, es decir **pasa a la siguiente iteración del bucle**.