Control Flow

Introducción al flujo de control

El flujo de control es la forma en que un programa decide qué instrucciones ejecutar y en qué orden. Gracias a él, un programa puede tomar decisiones, repetir tareas o seleccionar diferentes caminos según las condiciones que se presenten durante la ejecución. Sin control de flujo, un programa simplemente ejecutaría las instrucciones de arriba hacia abajo sin ninguna lógica adaptable.

Rust ofrece varias construcciones de control como: (if, loop, while, for y match) que permiten crear comportamientos dinámicos y seguros.

if Expressions

Las if expressions permiten tomar decisiones basadas en una condición lógica.

En Rust, if no es solo una estructura de control es una expresión, lo que significa que puede retornar un valor.

Sintaxis de un if expression:

```
if condicion {
    // rama verdadera
} else {
    // rama falsa
}
```

Ejemplo:

```
fn main() {
    let numero = 7;

    if numero > 5 {
        println!("El número es mayor que 5");
    } else {
        println!("El número es menor o igual a 5");
    }
}
```

if como expresión

Como if devuelve un valor, puedes asignarlo directamente a una variable:

```
let mensaje = if numero > 5 {
    "El numero 10 es mayor que 5"
} else {
    "El numero 10 es menor o igual a 5"
};

println!("{}", mensaje);  //El numero 10 es mayor que 5
```

If expressions anidados También puedes encadenar múltiples condiciones:

```
let numero = 15;

let resultado:&str = if numero < 0 {

    "Negativo"

} else if numero == 0 {
```

```
    "Cero"

} else {
    "Positivo"
};
println!("{}", resultado); //Positivo
```

Tipos de retorno en if expressions

Regla importante en Rust:

Todas las ramas de un if deben devolver el mismo tipo.

```
let x:i32 = 5;
let y:i32 = if x > 0 { 1 } else { -1 }; // ambas ramas son i32
```

Esto seria un error:

```
let x = 5;
let y = if x > 0 { 1 } else { "menor o igual a cero" };
// Error: tipos distintos (i32 vs &str)
```

If con bloques de código

Las ramas pueden contener bloques completos.

El valor retornado es la última expresión del bloque:

```
let x = 2;

let res = if x % 2 == 0 {
    println!("Es par");

    "par"     // valor devuelto
} else {
    println!("Es impar");

    "impar"
};

println!("Resultado: {}", res);
```

if con shadowing

```
let edad = 20;

let edad = if edad >= 18 {
    "adulto"
} else {
    "menor"
};

println!("{}", edad);
```

if dentro de expresiones más grandes

if es 100% una expresión, no solo un control de flujo.

```rust
let mensaje = format!(
    "Estado: {}",
    if x > 0 { "positivo" } else { "no positivo" }
);

println!("{}", mensaje);
```

Operadores lógicos

Rust soporta los operadores lógicos más comunes:

- `&&` : AND lógico.
- `||` : OR lógico.
- `!` : NOT negación.

```rust
let x = false;
let y = true;

if x && (2 > 1) {
  println!("No se ejecuta");
}

if y || (2 < 1) {
  println!("Se ejecuta sin evaluar 2 < 1");
}

if !x {
  println!("!x = true");
}
```

Bucles

En Rust, un bucle es una estructura de control de flujo que permite ejecutar repetidamente un bloque de código, siguiendo reglas estrictas del sistema de tipos, la seguridad en memoria y el control explícito del flujo.

Rust separa los bucles porque cada uno expresa una intención distinta, y el compilador puede razonar mejor sobre el código.

Loops

Un loop permite ejecutar un bloque de código repetidamente mientras se cumpla una condición o hasta que se indique explícitamente que debe detenerse.

Sintaxis básica:

```rust
loop {
    // código que se repite
}
```

loop repite el código dentro de `{}` de manera infinita hasta que ocurra un break.

```rust
let mut contador = 0;

loop {
    contador += 1;
    println!("Contador: {}", contador);

    if contador == 5 {
```

```
        break; // salimos del loop
    }
}
```

Explicación del codigo:

1. `loop {`

- Iniciamos un bucle infinito con `loop`.
- Este bloque se repetirá hasta que lo detengamos con un `break`.
- Cuando contador llega a 5, el bucle termina.

Resultado:

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
Fin del loop
```

loop con continue

'continue' permite saltar inmediatamente a la siguiente iteración del bucle.

```
let mut n = 0;

loop {
    n += 1;

    if n % 2 == 0 {
        continue; // saltamos los pares
    }

    println!("Número impar: {}", n);

    if n > 7 {
        break;
    }
}
```

1. - 'continue' hace que se ignore el resto del bloque y pase a la siguiente vuelta.
   - Por eso solo se imprimen los números impares.

Salida:

```
Número impar: 1
Número impar: 3
Número impar: 5
Número impar: 7
Número impar: 9
```

loop como expresión

En Rust, casi todo es una expresión, incluyendo loop.

Puedes usar 'break' para devolver un valor.

```rust
fn main() {
    let mut contador = 0;

    let resultado = loop {
        contador += 1;

        if contador == 10 {
            break contador * 2; // devuelve 20
        }
    };

    println!("El resultado es {}", resultado);
}
```

Etiquetas en bucles ‘label’

Un label en Rust es un identificador con nombre que se asocia a un bucle para referenciarlo explícitamente desde instrucciones como break o continue, lo que permite controlar explícitamente qué bucle se rompe o continúa, especialmente en bucles anidados.

Sintaxis básica:

```rust
'mi_bucle: loop {
    // ...
}
```

- Comienza con ‘'‘.
- Va antes del bucle.
- Puede tener cualquier nombre válido

Sin labels, break y continue solo afectan al bucle más interno.

Ejemplo sin label:

```rust
loop {
    loop {
        break; // rompe SOLO el loop interno
    }
}
```

Las etiquetas permiten controlar loops anidados indicando exactamente de cuál quieres salir o continuar.

Ejemplo con label:

```rust
fn main() {
    let numeros = vec![10, 20, 30, 40, 50];
    let mut indice = 0;

    'busqueda: loop {
        if indice >= numeros.len() {
            println!("Número no encontrado.");
            break 'busqueda;
        }

        let valor = numeros[indice];

        if valor == 30 {
```

```
            println!("¡Encontramos el número {} en la posición {}!", valor, indice);
            break 'busqueda;
        }

        indice += 1;
    }

    println!("Fin del programa.");
}
```

Explicación

- La etiqueta `'busqueda: loop` identifica este loop con un nombre.
- `break 'busqueda` permite salir específicamente de este bucle muy útil si hay loops dentro de loops.
- El programa busca el número 30 dentro del vector y rompe el bucle cuando lo encuentra.

Salida:

```
¡Encontramos el número 30 en la posición 2!
Fin del programa.
```

Ejemplo de loops anidados:

Este ejemplo muestra por qué existen las etiquetas: para poder romper bucles externos desde dentro de bucles internos.

```
fn main() {
    let mut fila = 0;
    let mut columna = 0;

    'outer: loop {       // loop externo
        println!("Fila {}", fila);

        'inner: loop {   // loop interno
            println!("  Columna {}", columna);

            if columna == 2 {
                break 'outer; // rompemos el loop externo
            }

            columna += 1;
        }

        fila += 1;
    }

    println!("Fin del programa.");
}
```

Explicación:

- ‘outer es el loop externo.
- ‘inner es el loop interno.
- En lugar de romper solo el loop interno, usamos break ‘outer para salir directamente del externo.

Resultado esperado:

```
Fila 0
  Columna 0
  Columna 1
  Columna 2
Fin del programa.
```

While

El bucle while repite un bloque de código mientras una condición booleana sea verdadera. Es útil cuando no sabes de antemano cuántas iteraciones habrá, pero sí tienes una condición que determina cuándo detenerte.

Sintaxis:

```
while condicion {
    // código que se ejecuta mientras condicion sea true
}
```

Ejemplo basico:

```
fn main() {
    let mut contador = 0;

    while contador < 5 {
        println!("contador = {}", contador);
        contador += 1; // importante: actualizar la condición en algún punto
    }
}
```

Explicación:

- Antes de cada iteración se evalúa contador < 5.
- Si es true, se ejecuta el bloque; si es false, se sale del while.
- Es responsabilidad del programador asegurarse de que la condición cambie (normalmente modificando variables dentro del bucle); de lo contrario el bucle será infinito.

Resultado:

```
contador = 0
contador = 1
contador = 2
contador = 3
contador = 4
```

while con break y continue

Dentro de un while puedes usar break para salir inmediatamente o continue para saltar al siguiente ciclo.

```
fn main() {
    let mut n = 0;

    while n < 10 {
        n += 1;

        if n % 2 == 0 {
            continue; // saltar el resto del bloque para los pares
        }
```

```
        println!("Número impar: {}", n);

        if n >= 7 {
            break; // salir del while cuando n >= 7
        }
    }
}
```

Explicación:

- continue salta a la evaluación de la condición (n < 10) sin ejecutar las líneas siguientes dentro del bloque.
- break termina el while inmediatamente.

Match
El match es una de las herramientas más poderosas del flujo de control en Rust. Permite tomar decisiones basadas en patrones, de una manera clara, segura y completamente exhaustiva. A diferencia de un simple if, match compara un valor contra múltiples patrones y obliga al programador a manejar todos los casos posibles.

Sintaxis:

```
match valor {

    Patrón1 => { /* código */ }

    Patrón2 => { /* código */ }

    _ => { /* código por defecto */ }

}
```

- valor : expresión a evaluar.
- Cada brazo (=>) representa un caso.
- El patrón _ es el caso “cualquiera”; actúa como un else.

Ejemplo basico:

La forma más simple de match consiste en comparar un valor con varias alternativas:

```
fn main() {
    let numero = 3;

    match numero {
        1 => println!("Uno"),
        2 => println!("Dos"),
        3 => println!("Tres"),
        _ => println!("Otro número"),
    }
}
```

Resultado:

```
Tres
```

Patrones

Los patrones permiten describir de forma declarativa qué valores queremos capturar o comparar. Rust soporta patrones muy expresivos:

Patrón literal

```
match x {
    0 => println!("Cero"),
    5 => println!("Cinco"),
    _ => println!("Otro valor"),
}
```

Patrón con múltiples alternativas

```
match x {
    1 | 2 | 3 => println!("Uno, dos o tres"),
    _ => println!("Otro"),
}
```

Patrones con variables

```
match x {
    10 => println!("Diez"),
    otro => println!("Otro valor: {}", otro),
}
```

Rangos

Rust permite usar rangos como patrones. Esto funciona tanto para enteros como para caracteres:

```
let edad = 20;

match edad {
    0..=12 => println!("Niño"),
    13..=17 => println!("Adolescente"),
    18..=59 => println!("Adulto"),
    _ => println!("Mayor de edad"),
}
```

Resultado:

```
Adulto
```

Destructuración simple

match también puede abrir o “desestructurar” datos, como tuplas o enums simples.

```
fn main() {
    let punto = (0, 5);

    match punto {
        (0, y) => println!("Está en el eje Y, valor: {}", y),
        (x, 0) => println!("Está en el eje X, valor: {}", x),
        (x, y) => println!("En coordenadas ({}, {})", x, y),
    }
}
```

Resultado:

```
Está en el eje Y, valor: 5
```

Condiciones lógicas

```rust
fn main() {
    let numero:i8 = 8;
    match numero {
        value if value % 2 == 0 => println!("El numero {numero} es par"),
        value if value % 2 == 1 => println!("El numero {numero} es impar"),
         _ => unreachable!(), // aquí decimos que nunca debería entrar
    }
}
```

Aquí, el patrón `value if ...` nos permite agregar condiciones adicionales dentro del match.

match como expresión

En Rust, match devuelve un valor, igual que un if expresión. Esto permite asignar el resultado:

```rust
let numero = 2;

let texto = match numero {
    1 => "uno",

    2 => "dos",

    3 => "tres",

    _ => "desconocido",
};

println!("Número: {}", texto);
```

Esto vuelve a match más poderoso que en muchos otros lenguajes, pues no solo ejecuta código: produce valores.

For
El bucle `for` es la forma más idiomática y segura de iterar en Rust.

Rust no usa índices manuales como en otros lenguajes, sino que itera directamente sobre colecciones, lo que evita errores como desbordes, índices fuera de rango o ciclos infinitos.

For sobre rangos

La forma más común de usar un `for` es con rangos `(start..end)`:

```rust
for i in 0..5 {
    println!("i = {}", i);
}
```

Salida:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Notas importantes sobre rangos:

- `a..b` incluye `a` pero excluye `b`
- Para incluir el final, usa `a..=b`:

```rust
for n in 1..=3 {
    println!("{}", n);
}
```

Salida:

```
1
2
3
```

Los rangos se convierten automáticamente en iteradores, por eso funcionan con `for`.

For sobre arrays

Puedes iterar directamente sobre arrays:

```rust
let numeros = [10, 20, 30, 40, 50];

for numero in numeros {
    println!("Número: {}", numero);
}
```

Salida:

```
Número: 10
Número: 20
Número: 30
Número: 40
Número: 50
```

Iterar con referencia:

Si no quieres mover los valores, usa &:

```rust
let palabras = ["Hola", "Mundo", "Rust"];

for palabra in &palabras {
    println!("{}", palabra);
}

// palabras sigue disponible después del loop
println!("Primera palabra: {}", palabras[0]);
```

For sobre vectores

Los vectores se iteran igual que los arrays:

```rust
let mut puntos = vec![100, 200, 300, 400];

for punto in &puntos {
    println!("Punto: {}", punto);
}
```

Salida:

```
Punto: 100
Punto: 200
Punto: 300
Punto: 400
```

Iterar y modificar elementos:

Para modificar elementos durante la iteración, usa `&mut`:

```
let mut numeros = vec![1, 2, 3, 4, 5];

for numero in &mut numeros {
    *numero *= 2;  // multiplica cada elemento por 2
}

println!("{:?}", numeros);  // [2, 4, 6, 8, 10]
```

Explicación:
- `&mut numeros` da acceso mutable a cada elemento
- `*numero` desreferencia para modificar el valor original

For sobre strings

Iterar sobre caracteres:

```
let texto = "Hola";

for caracter in texto.chars() {
    println!("{}", caracter);
}
```

Salida:

```
H
o
l
a
```

Iterar sobre bytes:

```
let texto = "Rust";

for byte in texto.bytes() {
    println!("{}", byte);
}
```

Salida:

```
82
117
115
116
```

⚠ Para texto normal, usa `.chars()`. Los bytes son útiles para procesamiento de bajo nivel.

Iterar sobre palabras:

```
let frase = "Hola desde Rust";

for palabra in frase.split_whitespace() {
    println!("{}", palabra);
}
```

Salida:

```
Hola
desde
Rust
```

Iterar sobre líneas:

```
let texto = "Primera línea\nSegunda línea\nTercera línea";

for linea in texto.lines() {
    println!("Línea: {}", linea);
}
```

Salida:

```
Línea: Primera línea
Línea: Segunda línea
Línea: Tercera línea
```

For con índice

Si necesitas el índice además del valor, usa `.enumerate()`:

```
let colores = ["Rojo", "Verde", "Azul"];

for (indice, color) in colores.iter().enumerate() {
    println!("Color {} es {}", indice, color);
}
```

Salida:

```
Color 0 es Rojo
Color 1 es Verde
Color 2 es Azul
```

Con vectores:

```
let numeros = vec![10, 20, 30];

for (i, numero) in numeros.iter().enumerate() {
    println!("numeros[{}] = {}", i, numero);
}
```

Salida:

```
numeros[0] = 10
numeros[1] = 20
numeros[2] = 30
```

Break en for

Puedes salir de un `for` anticipadamente con `break`:

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for numero in numeros {
    if numero == 5 {
        println!("Encontramos el 5, detenemos la búsqueda");
        break;
    }
    println!("Número: {}", numero);
}
```

Salida:

```
Número: 1
Número: 2
Número: 3
Número: 4
Encontramos el 5, detenemos la búsqueda
```

Continue en for

Salta a la siguiente iteración con `continue`:

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for numero in numeros {
    if numero % 2 == 0 {
        continue;  // saltar números pares
    }
    println!("Número impar: {}", numero);
}
```

Salida:

```
Número impar: 1
Número impar: 3
Número impar: 5
Número impar: 7
Número impar: 9
```

For anidados

Puedes anidar bucles `for`:

```
for fila in 1..=3 {
    for columna in 1..=3 {
        println!("Posición ({}, {})", fila, columna);
    }
}
```

Salida:

```
Posición (1, 1)
Posición (1, 2)
Posición (1, 3)
Posición (2, 1)
Posición (2, 2)
Posición (2, 3)
Posición (3, 1)
Posición (3, 2)
Posición (3, 3)
```

Ejemplo práctico: tabla de multiplicar

```
for i in 1..=5 {
    for j in 1..=5 {
        print!("{:3} ", i * j);
    }
    println!();  // nueva línea después de cada fila
}
```

Salida:

```
  1   2   3   4   5
  2   4   6   8  10
  3   6   9  12  15
  4   8  12  16  20
  5  10  15  20  25
```

Iterar sobre arrays multidimensionales

```rust
let matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
];

for fila in matriz {
    for elemento in fila {
        print!("{} ", elemento);
    }
    println!();
}
```

Salida:

```
1 2 3
4 5 6
7 8 9
```

Iterar en reversa

Usa `.rev()` para invertir un rango:

```rust
for i in (1..=5).rev() {
    println!("{}", i);
}
```

Salida:

```
5
4
3
2
1
```

Con arrays:

```rust
let numeros = [10, 20, 30, 40, 50];

for numero in numeros.iter().rev() {
    println!("{}", numero);
}
```

Salida:

```
50
40
30
20
10
```

Iterar con pasos (step)

Para iterar con saltos, usa `.step_by()`:

```
for i in (0..10).step_by(2) {
    println!("{}", i);
}
```

Salida:

```
0
2
4
6
8
```

Casos comunes de for

Suma de elementos:

```
let numeros = [1, 2, 3, 4, 5];
let mut suma = 0;

for numero in numeros {
    suma += numero;
}

println!("Suma total: {}", suma);  // 15
```

Encontrar máximo:

```
let numeros = [5, 2, 8, 1, 9, 3];
let mut maximo = numeros[0];

for numero in numeros {
    if numero > maximo {
        maximo = numero;
    }
}

println!("Máximo: {}", maximo);  // 9
```

Contar elementos que cumplen condición:

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let mut pares = 0;

for numero in numeros {
    if numero % 2 == 0 {
        pares += 1;
    }
}

println!("Números pares: {}", pares);  // 5
```

Buscar elemento:

```
let nombres = ["Ana", "Luis", "Carlos", "María"];
let buscar = "Carlos";
let mut encontrado = false;

for nombre in nombres {
```

```
    if nombre == buscar {
        encontrado = true;
        break;
    }
}

if encontrado {
    println!("{} está en la lista", buscar);
} else {
    println!("{} no está en la lista", buscar);
}
```

Diferencias importantes

for vs while con índice manual:

✖ Forma propensa a errores (estilo C):

```
let arr = [10, 20, 30];
let mut i = 0;

while i < arr.len() {
    println!("{}", arr[i]);
    i += 1;
}
```

✔ Forma idiomática de Rust:

```
let arr = [10, 20, 30];

for elemento in arr {
    println!("{}", elemento);
}
```

Ventajas del `for` idiomático:

- No puedes olvidar incrementar el índice
- No puedes acceder fuera de rango
- Más legible y conciso
- El compilador puede optimizar mejor

Consumir vs prestar

Consumir (mover):

```
let vec = vec![1, 2, 3];

for item in vec {  // vec se mueve
    println!("{}", item);
}

// println!("{:?}", vec);  // ✗ Error: vec fue movido
```

Prestar (con referencia):

```
let vec = vec![1, 2, 3];

for item in &vec {  // vec se presta
    println!("{}", item);
}
```

```
println!("{:?}", vec);  // ✓ vec sigue disponible
```

Modificar (con referencia mutable):

```
let mut vec = vec![1, 2, 3];

for item in &mut vec {
    *item *= 2;
}

println!("{:?}", vec);  // [2, 4, 6]
```

Resumen de for

```
// Rangos
for i in 0..5 { }
for i in 1..=10 { }
for i in (0..10).rev() { }
for i in (0..10).step_by(2) { }

// Arrays
for item in arr { }
for item in &arr { }

// Vectores
for item in vec { }
for item in &vec { }
for item in &mut vec { *item = ...; }

// Strings
for c in s.chars() { }
for b in s.bytes() { }
for palabra in s.split_whitespace() { }
for linea in s.lines() { }

// Con índice
for (i, item) in arr.iter().enumerate() { }

// Control
break;     // salir del loop
continue;  // siguiente iteración
```