


VNCKey

 RUST PERÚ

Diviértete con Rust

Viaja al futuro con el poder de Rust



 Programación de
Sistemas
⚡ Zero-Cost
Abstractions
🔒 Memory Safety

Alexander
Villanueva

Contenido

Introducción a Rust	4
Instalación de Rust	5
Compilador y Gestor de Proyectos	5
Tu Primer Proyecto con Cargo	7
Variables con Rust	9
Sistema de Memoria	14
Tipos de datos	16
Scalar Types	17
Enteros	17
Punto Flotante	22
Booleano	25
Carácter	29
Compound Types	31
Tuplas	31
Arrays	34
Collections	40
Vectores	40
Strings	51
HashMap	66
Control flow	83
Control Flow	83
if Expressions	84
Bucles	86
Loops	87
While	91
Match	92
For	95
Iteradores	105
Funciones en Rust	116
Struct	123
Sintaxis básica	124
Instanciación de un Struct	124
Acceso a los campos	124
Structs Mutables	125
Struct Update Syntax	125
Tuple Structs	125
Unit-like Structs	125
Desestructuración de Structs	126
Derivaciones automáticas	126
Métodos en Structs	126
Self	127
Method Chaining	130
Enums y Patrones	131
Generic Types	134
Traits	135
Trait Bounds	140
Lifetimes	141

Manejo de Errores	145
Errores Recuperables	145
Option<T>	145
Result<T, E>	146
El operador ?	147
Errores NO Recuperables	148
panic!	148
Backtrace	149

Introducción a Rust

Durante más de cinco décadas, los lenguajes de programación de bajo nivel como C y C++ han sido la base sobre la cual se construyeron componentes críticos de la computación moderna, incluyendo sistemas operativos, kernels, navegadores y software de alto rendimiento. Su principal fortaleza radica en el control preciso que ofrecen sobre la memoria y el hardware, lo que permitió alcanzar niveles de eficiencia difíciles de igualar por otros lenguajes.

Sin embargo, esta misma libertad expuso a los desarrolladores a una realidad compleja y costosa. La gestión manual de memoria y la falta de garantías de seguridad en el propio lenguaje facilitaron la introducción accidental de errores graves como buffer overflow, use-after-free, double free y data races. Estos problemas no solo son difíciles de detectar, sino que han sido históricamente responsables de una gran parte de las vulnerabilidades de seguridad más críticas en la industria del software.

Frente a este panorama, Graydon Hoare un brillante Ingeniero con amplia experiencia en la corrección de errores en C y C++ formuló una pregunta tan simple como disruptiva.

¿Por qué los lenguajes de programación permiten errores que, conceptualmente, no deberían ser posibles?

De esta reflexión nació Rust, un lenguaje de programación moderno diseñado desde sus cimientos para prevenir clases enteras de errores antes de que el programa llegue siquiera a ejecutarse.

Rust introduce un modelo de seguridad de memoria verificado en tiempo de compilación, sin depender de recolectores de basura del cual se sacrifica rendimiento. El resultado es un lenguaje capaz de ofrecer un desempeño comparable al de C y C++, pero con garantías mucho más sólidas en términos de seguridad, concurrencia y corrección.

¿Por qué las empresas están apostando por Rust?

La adopción de Rust en la industria no es una moda pasajera, sino una respuesta directa a problemas reales y costosos. Grandes empresas tecnológicas han comprobado que una proporción significativa de sus vulnerabilidades de seguridad provienen de errores de memoria. Reducir estos fallos no solo mejora la seguridad, sino que también disminuye costos de mantenimiento, parches y auditorías.

Empresas como Mozilla, Microsoft, Google, Amazon y Cloudflare han incorporado Rust en componentes críticos de sus sistemas. En muchos casos, Rust se ha utilizado para reescribir partes sensibles del código originalmente escritas en C o C++, logrando mejoras sustanciales en seguridad sin perder rendimiento.

Por ejemplo:

- Mozilla desarrolló partes del motor de Firefox en Rust para reducir fallos de memoria.
- Microsoft ha promovido activamente Rust como alternativa segura para el desarrollo de componentes del sistema operativo.
- Google ha integrado Rust en Android y en proyectos de infraestructura donde la seguridad es prioritaria.
- Cloudflare utiliza Rust en servicios de red de alto rendimiento expuestos directamente a Internet.

Estos casos de éxito demuestran que Rust no solo es un lenguaje académico o experimental, sino una herramienta probada en producción a gran escala.

Instalación de Rust

Rust se instala mediante `rustup`, el instalador oficial que gestiona versiones del compilador y herramientas asociadas.

Linux y macOS

Ejecuta en tu terminal:

```
curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Esto instalará:

- El compilador: `rustc`
- Gestor de paquetes: `Cargo`
- Documentación estándar
- Herramientas de desarrollo

Presiona `Enter` para la instalación estándar. Luego reinicia tu terminal o ejecuta:

```
source $HOME/.cargo/env
```

Windows

Requisito previo: Visual Studio Build Tools con MSVC

Paso 1: Instalar Build Tools

Descarga desde: <https://visualstudio.microsoft.com/es/downloads/>

Tienes 2 Opciones:

- **Opción completa:** Workload “Desktop Development with C++”
- **Opción mínima:** En “Componentes individuales” selecciona:
 - MSVC v143 – VS 2022 C++ x64/x86 build tools
 - Windows 11 SDK

Paso 2: Instalar Rust

Descarga desde: <https://www.rust-lang.org/tools/install>

Selecciona el instalador según tu arquitectura `32-bit`, `x64`, o `ARM64`.

- Ejecuta **`rustup-init.exe`** y presiona `Enter` para instalación estándar.
- Reinicia tu terminal después de la instalación.

Verificación

Confirma que la instalación fue exitosa:

```
rustc --version # rustc 1.89.0 (29483883e 2025-08-04)
cargo --version # cargo 1.89.0 (c24e10642 2025-06-23)
rustup --version # rustup 1.27.0 (2024-08-14)
```

Compilador y Gestor de Proyectos

Un lenguaje de programación moderno no se define únicamente por su sintaxis, sino también por las herramientas que lo rodean. En Rust, el compilador y su gestor de proyectos forman parte esencial del ecosistema y están diseñados para trabajar de manera integrada. Comprender su función desde el inicio permite entender por qué Rust ofrece una experiencia de desarrollo distinta a la de muchos otros lenguajes.

El compilador de Rust

En muchos lenguajes, el compilador se limita a traducir el código fuente a un programa ejecutable y a señalar errores básicos de sintaxis. En Rust, en cambio, el compilador asume un papel mucho más activo, ya que su propósito es ayudar al programador a escribir código correcto, seguro y eficiente.

Además de verificar el código, el compilador de Rust transforma los archivos fuente con extensión `.rs` en ejecutables nativos o en bibliotecas reutilizables, según el tipo de proyecto. Este proceso puede aplicarse tanto a proyectos completos como a archivos simples, lo que permite compilar desde pequeños programas de prueba hasta aplicaciones y bibliotecas de gran escala.

Para ello, el compilador analiza no solo la estructura del programa, sino también el uso de la memoria, los tiempos de vida de los datos y el acceso concurrente a los recursos. Gracias a este enfoque, Rust puede detectar en tiempo de compilación errores que en otros lenguajes suelen aparecer únicamente en ejecución o en producción.

Aunque esta rigurosidad puede resultar exigente al inicio, cumple una función pedagógica fundamental y es enseñar a pensar correctamente sobre la memoria y la seguridad desde el primer día. A esto se suma la calidad de los mensajes de error del compilador, que no solo indican qué está mal, sino que explican por qué ocurre el problema y cómo corregirlo, convirtiéndose en una herramienta de aprendizaje continuo.

Proceso de Compilación



Diagrama 1: Flujo de compilación en Rust: desde el código fuente hasta el ejecutable

Como vemos en el Diagrama 1, el compilador `rustc` es el encargado de transformar nuestro código.

Todo comienza con el código fuente `main.rs`:

```
fn main() {  
    println!("Compilador directo rustc."); //Archivo: main.rs  
}
```

🦀 Rust

- `fn`: Define la función principal y obligatoria de un programa ejecutable en Rust.
- `main()`: Es un nombre reservado y especial que el compilador de Rust y el sistema operativo buscan para saber dónde empezar a ejecutar el código.

Podemos compilarlo:

```
rustc main.rs
```

🐚 bash

Esto genera un ejecutable:

- Linux/macOS: `main`
- Windows: `main.exe`


Ejecuta:

```
./main          # Linux/macOS  
main.exe        # Windows
```

🐚 bash

Resultado:

```
Ho!a desde Rust!
```

 Output

Gestor de Proyectos Cargo

Junto al compilador, Rust incorpora de forma nativa Cargo, su gestor de proyectos y paquetes. Cargo se encarga de tareas fundamentales como la creación de proyectos, la compilación del código, la gestión de dependencias, la ejecución de pruebas y la generación de documentación, todo bajo una estructura clara y coherente.

Para quienes se inician en la programación, Cargo elimina gran parte de la complejidad asociada a la configuración del entorno y al manejo manual de librerías. Para los usuarios avanzados, ofrece control preciso de versiones, reproducibilidad y una integración fluida con el ecosistema de bibliotecas de Rust, disponibles a través del repositorio central crates.io.

Fases del Proceso

1. Paso 1: Creación del Módulo Fuente

Todo comienza con el código fuente, que tradicionalmente lleva la extensión `.rs`.

```
fn A main B () {  
    println!("Compilador directo rustc."); //Archivo: main.rs  
}
```

 Rust

2. Paso 2: Compilación

Desde la terminal, se invoca a rustc, apuntando al archivo de entrada. El compilador lee el código y genera un archivo binario ejecutable en el mismo directorio.

```
rustc main.rs
```

 bash

En este proceso, rustc maneja internamente la verificación de tipos, el borrow checker y la generación del código máquina optimizado, utilizando LLVM.

3. Ejecución

Esto genera un ejecutable.

- Windows:

```
.\main.exe
```

 bash


- Linux/macOS:

```
./main
```

 bash

Resultado:

```
Compilador directo rustc.
```

 Output

Tu Primer Proyecto con Cargo

Crear un nuevo proyecto

```
# Crear un proyecto binario (aplicación)
```

 bash

```
cargo new a hola_mundo b
```


- a: Crear un nuevo proyecto Rust
- b: Nombre del proyecto

```
# Entrar al directorio
cd hola_mundo
```

 bash

Estructura creada:

```
hello_world/ Raíz del proyecto
├── Cargo.lock Registra las versiones específicas
├── Cargo.toml Define las dependencias
├── src/
│   └── main.rs Código fuente principal
└── target/ Destino de la Compilación
    ├── debug/
    │   ├── build/
    │   └── deps/
    └── hello_world El ejecutable de tu aplicación
```

 Output

Anatomía del Proyecto: Cargo.toml

Contenido inicial de Cargo.toml

```
[package]
name = "hola_mundo" Nombre del proyecto
version = "0.1.0" Versión siguiendo
edition = "2021" Edición estable de Rust 2021

[dependencies] crates
# Ejmplos
# rand = "0.8.5" Permite generar números aleatorios
# serde = "1.0.130" Permite serializar y deserializar datos

[dev-dependencies]
# Dependencias solo para desarrollo/testing

[build-dependencies]
# Dependencias para scripts de build
```


 toml

} (1)
Metadatos

} (2)
Librerías externas

Punto de entrada: main.rs


```
fn main() A { C
    println! B ("Hola, Rust!");
}
```

 Rust

1. A: Define la función principal del programa
2. B: Es una macro que imprime texto en la consola
3. C: Delimitan el bloque de código de la función

Compilar y Ejecutar

```
cargo run
```

 bash

```
Compiling hola_mundo v0.1.0 (/ruta/hola_mundo) A
Finished `dev` profile [unoptimized + debuginfo] target(s) in 3.42s B
Running `target/debug/hola_mundo` C
Hola, Rust! D
```

 bash

1. A: Cargo compila el proyecto (solo la primera vez o si hay cambios)
2. B: Perfil de compilación: dev (desarrollo, sin optimizaciones)
3. C: Ruta del ejecutable que se está ejecutando
4. D: Output de tu programa

Si no modificaste el código, la segunda ejecución será instantánea:


```
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/hola_mundo`
Hola, Rust!
```

 bash

Variables con Rust


Let

```
fn main() {
    let A edad B = 25 C;
    println!("Mi edad es {}", edad);
}
```

 Rust

1. A: La palabra reservada let se usa para declarar variables inmutables por defecto
2. B: Nombre de la variable edad
3. C: Asignación de tipo de valor entero 25

```
Mi edad es 25
```

 Output

Que pasa si queremos modificar el valor de una variable inmutable?

```
fn main() {
    let edad = 25;
    edad = 26; ERROR!!
```

 Rust

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

Gracias a herramientas inteligentes como rust-analyzer y rustc, nuestros editores de código pueden analizar información avanzada e interactiva a través del Language Server Protocol (LSP). De esta manera, es posible visualizar errores, comprender por qué el código es incorrecto e incluso recibir sugerencias automáticas para corregirlo.

```
error[E0384]: No se puede asignar dos veces a la variable inmutable `edad`
--> src/main.rs:3:3
3 |     edad = 26;
  |
  | help: consider making this binding mutable
2 |     let mut edad = 25;
  |         +++
For more information about this error, try `rustc --explain E0384`.
```

1. El Código de Error: [E0384]

Este es el identificador único y universal del problema.

Con documentacion https://doc.rust-lang.org/error_codes/error-index.html exacta para el tipo de error y explicacion detallada de ese problema.

variables mutables

Si necesitas cambiar el valor de una variable, debes declararla explícitamente como mutable usando `let mut`.

```
fn main() {
    let mut A carro = "Toyota";
    println!("Mi carro es {}", carro);
    carro = "Honda";
    println!("Mi nuevo carro es {}", carro);
}
```

1. A: La palabra reservada `let mut` se usa para declarar variables mutables que cambian su valor a lo largo del programa.

Shadowing


El shadowing permite declarar una nueva variable con el mismo nombre que una anterior.

La nueva variable “sombrea” a la anterior.

```
fn main() {
    let mut edad = 25;
    println!("Mi edad es {}", edad);
    let edad = "Mi edad es 35"; A
    println!("{}", edad);
}
```


- A:
 - Rust permite declarar una nueva variable con el mismo nombre que una anterior.
 - La nueva variable “sombrea” a la anterior.
 - Shadowing permite cambiar el tipo de una variable.

```
Mi edad es 25
Mi edad es 35
```

 Output

Lo que no se puede hacer es cambiar el tipo de una variable sin “sombrear”.

```
let mut texto = "Hola";
texto = 5; ERROR!
```

 Rust

Scopes

El scope determina dónde una variable es válida en tu código. En Rust, el scope está definido por llaves {}.


```
fn main() {
    let x = 5;
    println!("Valor de x: {x}");
    {
        let x = x * 2;
        let y = x;
        println!("Dentro del scope x: {x}");
        println!("Dentro del scope y: {y}");
    } A
    //println!("Valor de y: {y}"); ERROR!
    println!("Valor de x: {x}");
}
```

 Rust

- A:
 - La variable x y y son válidas dentro del scope en el que fueron declaradas.
 - Terminado el scope, las variables x y y son liberadas.
 - Ya no se pueden usar fuera del scope en el que fueron declaradas.

Nota: Puedes crear scopes anidados.

```
Valor de x: 5
Dentro del scope x: 10
Dentro del scope y: 10
Valor de x: 5
```

 Output

Constantes

Las constantes son valores globales que nunca cambian y deben tener un tipo explícito. No tienen una dirección de memoria fija. Se utiliza para valores que son absolutamente fijos y conocidos de antemano, como constantes matemáticas, límites, o configuraciones fijas.

```
const PI A : f64 B = 3.14159265359;
```

 Rust

```
fn main() {  
    const PI: f64 = 5.14; C  
    println!("Valor de PI: {}", PI);  
    //PI = 3.12; D  
}
```

1. A:

Las constantes siempre usan SCREAMING_SNAKE_CASE.

2. B:

El tipo debe ser explícito :f64 en este caso flotante.


3. C:

Rust permiten sombrear constantes con el mismo nombre.

4. D:

Rust no permite mutar constantes.

Valor de PI: 5.14

 Output

Valores estaticos

Las variables estáticas tienen una ubicación fija en memoria y viven durante toda la ejecución del programa. Se inicializan al inicio de la ejecución del programa (cuando el programa se carga, antes de que se ejecute la función main).

```
static PROTOCOLO_VERSION A : u8 = 2;
```

 Rust

```
fn main() {  
    // let PROTOCOLO_VERSION:u8 = 3; B  
    // PROTOCOLO_VERSION: u8 = 8; C  
    println!("Protocolo v{}", PROTOCOLO_VERSION);  
}
```

1. A:

Declaramos un valor estatico con static


2. B

No podemos sombrear un valor estatico con el mismo nombre.

3. C

No podemos mutar un valor estatico.

Protocolo v2

 Output

Statements & Expressions

Una sentencia es una instrucción que realiza una acción y no devuelve un valor. En Rust, la mayoría de las sentencias terminan con un punto y coma ;.

Una expresión es cualquier pieza de código que se evalúa y devuelve un valor.

```
fn main() {
    let y = { A
        let z = 3; B
        z + 1 D
    }; A

    println!("y = {}", y); A
}
```

 Rust

1. A:

- Tenemos la primera Sentencia (statement) `let y = { ... };`, una unidad de ejecución que no produce un valor que pueda ser utilizado por otra parte del código.
- Expresión Asignada: `{ ... }` La expresión de bloque se evalúa y devuelve un valor

2. B:

- Tenemos la segunda Sentencia (statement) La sentencia `let` realiza la acción de vincular un valor a un nombre y nunca devuelve un valor, Rust evita side-effect oculto.


```
let y = (let x = 5); ERROR!!!!
```

Ejemplos:

Javascript


```
let x = 1;
let y = (x = 2, x++); // y = 2; x = 3
console.log(y, x);    // 2 3 = side-effect dentro de la expresión

if (count = 0) { }    // 0 es falsy = nunca entra, pero *asigna*
```

 JavaScript

Python

```
b = 5
a = (b := 1) + (b := 2) # a = 2
print("Valor de a es: ",a) #Valor de a es: 3
print("Valor de b es: ",b) #Valor de b es: 2
```

 Python

- Rust prohíbe que `let` devuelva valor y así evita bugs clásicos como `if (x = 5)`.
- `println!` devuelve `unit type ()`, y la llamada como declaración.
- Rust prohíbe que `let` sea una expresión para eliminar una clase entera de errores que sí existen en lenguajes donde la asignación devuelve valor.
- La regla de oro en Rust es que la mayoría de las sentencias terminan con un ;.

3. C:

- Tenemos la primera Expresión (Expression) Cuando omites el punto y coma en la última línea de un bloque, le estás diciendo al compilador:


“Quiero que el valor resultante de esta operación sea el valor de retorno de todo el bloque.”

4. D:

Por ultimo, tenemos un Statement


Aunque la llamada a la macro `println!` es técnicamente una expresión (ya que se evalúa), su valor de retorno es el tipo unitario `()` (pronunciado “unit”).

```
y = 4
```

 Output

En caso de poner `;` al final se convierte en una sentencia (statement) y devuelve un unit type `()`.


```
fn main() {
    let y = {
        let z = 3;
        z + 1; A
    };
    println!("y = {:?}", y);
}
```

 Rust

1. A:

Devuelve unit type `()`

```
y = ()
```

 Output

Sistema de Memoria

La seguridad de memoria de Rust garantiza que un programa nunca acceda a memoria inválida ni cometa errores peligrosos al manejar recursos. En lenguajes como C y C++, esta responsabilidad recae por completo en el programador, lo que abre la puerta a introducir accidentalmente vulnerabilidades como desbordamientos de búfer, use-after-free, double free o data races.

Rust elimina estos problemas desde su diseño.

Todo esto sin la necesidad de un garbage collector y sin costo adicional en tiempo de ejecución. Este enfoque permite escribir software seguro y eficiente.


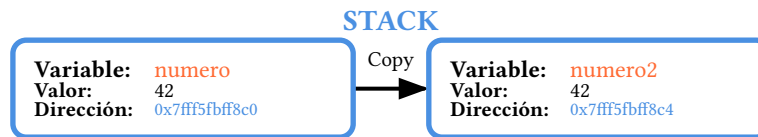
Los primeros conceptos pilares es comprender como se administra la memoria. Rust divide la memoria en dos grandes regiones Stack y Heap, su diferencia es esencial para evitar errores como:

- use-after-free
- double free
- dangling pointers
- data races

Stack

El Stack es una región de la memoria RAM para almacenar datos cuya vida útil y tamaño son conocidos en tiempo de compilación.

```
fn main() {
    let numero: u8 = 5;
    let numero2 = numero;
}
```

 Rust


Los valores se copian porque `u8` implementa `Copy`


Diagrama 2: Representación del stack de memoria en Rust

Heap

El Heap es una región de la memoria RAM para almacenar datos cuya vida útil y tamaño son conocidos en tiempo de ejecución.

```
fn main() {
    let texto : String A = String::from("Rûst"); B
    let texto2: String = texto; C

    println!("{texto2}");
    //println!("{texto1}"); D
}
```

 Rust

1. A:

- String es un tipo de dato que se almacena en el Heap y su tamaño es desconocido en tiempo de compilación.
- Se usa para representar texto dinámico, cuya longitud o contenido puede cambiar en tiempo de ejecución.

2. B:

- Sirve para construir una String a partir de un `&str` por ejemplo un "Hola" u otros tipos convertibles, como otro mismo String.
- Es equivalente a `.to_string()` en muchos casos, pero se prefiere `String::from()` por ser más explícito y genérico.

3. C:

- texto tiene un comportamiento llamado Move cuando texto2 la consume.
- Después de la asignación, texto ya no es válido, puesto que el compilador no te dejará usarlo, move semantics.
- Primera regla de Ownership solo puede haber un único propietario y es por esta razón que Rust no permite que dos variables apunten al mismo dato en el Heap.
- texto2 es ahora el propietario del String "Rûst"

- `println!("{}", texto);`

4. D:

- El uso de la variable `texto` es inválido, ya que fue movida a `texto2`.

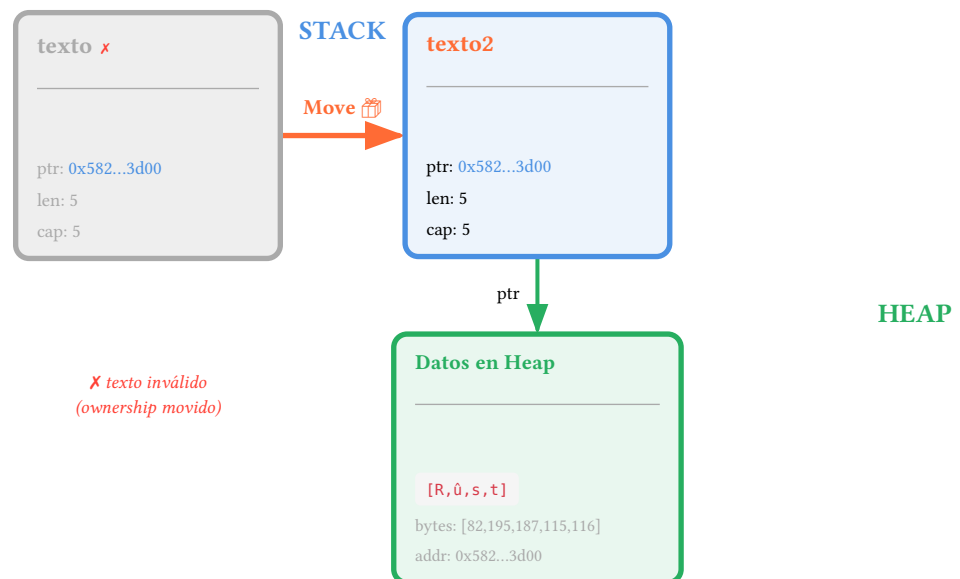


Diagrama 3: Move semántico: el ownership se transfiere de `texto` a `texto2`

Tipos de datos

Los tipos de datos son uno de los pilares fundamentales de cualquier lenguaje de programación, y Rust no es la excepción. En esencia, un tipo de dato define qué clase de información puede almacenar una variable o una expresión, así como las operaciones que se pueden realizar sobre ella.

Rust, al ser un lenguaje con tipado estático, requiere que el compilador conozca el tipo de cada valor en tiempo de compilación. Esto significa que los tipos se verifican antes de que el programa se ejecute, lo que permite detectar muchos errores potenciales de manera temprana.

¿Por qué son tan importantes los tipos en Rust?

La respuesta radica en sus principios del lenguaje mismo.

- **Seguridad de memoria y de tipos:** Rust utiliza su sistema de tipos para prevenir clases enteras de errores comunes en otros lenguajes, como el acceso a memoria no inicializada, desbordamientos de búfer o el uso incorrecto de valores nulos. Gracias al tipo estricto, el compilador puede garantizar que tu código no realice operaciones inválidas sin necesidad de un recolector de basura o de verificaciones en tiempo de ejecución costosas.
- **Rendimiento:** Al conocer los tipos en tiempo de compilación, Rust puede optimizar el código generado de forma muy eficiente. Cada tipo tiene un tamaño y un alineamiento conocidos, lo que permite al compilador generar código máquina cercano al que escribirías en C o C++, pero con muchas más garantías de seguridad.
- **Legibilidad y mantenibilidad:** Al declarar explícitamente los tipos, el código estará mejor documentado y entender rápidamente qué tipo de datos se está manipulando y qué comportamiento se espera.

En Rust, los tipos se dividen en categorías:

- Tipos escalares “scalar types”, en esta categoría representa un único valor.

- Tipos compuestos “compound types”, pueden agrupar múltiples valores en una sola estructura.

Scalar Types

Los tipos escalares en Rust representan un único valor básico. Son los bloques de construcción más simples del lenguaje y cubren los datos más comunes como los números enteros, números de punto flotante, valores booleanos y caracteres.

Rust es estricto con los tipos escalares, ya que cada uno tiene un tamaño definido por defecto, pero también te permite elegir variantes con tamaños explícitos según las necesidades.

Enteros

Los enteros son tipos de datos numéricos que representan valores sin parte decimal. Están diseñados para ser explícitos tanto en su tamaño como en su signo, proporcionando un control preciso sobre el uso de memoria y garantizando un comportamiento predecible del programa.

Clasificación de los Enteros

Rust organiza los tipos enteros en dos categorías principales según cual se adapte a las necesidades en términos de rango de valores y consumo de memoria.

Enteros con Signo

Los enteros con signo son aquellos tipos enteros que pueden representar tanto valores positivos como negativos, además del cero. En Rust, se identifican por el prefijo `i` seguido del número de bits: `i8`, `i16`, `i32`, `i64` y `i128`.

Tipo	Valor mínimo	Valor máximo
<code>i8</code>	-128	127
<code>i16</code>	-32 768	32 767
<code>i32</code>	-2 147 483 648	2 147 483 647
<code>i64</code>	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
<code>i128</code>	-2^{127}	$2^{127} - 1$

Tabla 1: Rangos numéricos de los tipos enteros con signo

El número en el nombre del tipo indica la cantidad de bits utilizados para almacenar el valor. Por ejemplo, `i8` utiliza 8 bits, mientras que `i128` utiliza 128 bits.

```
let temperatura: i8 = -15;
let poblacion: i32 = -2_147_483_648;
let deuda: i64 = -9_223_372_036_854_775_808;
```

Rust

Enteros sin Signo (`u`)

Los enteros sin signo representan exclusivamente valores no negativos (positivos y cero). Al no reservar un bit para el signo, pueden almacenar números positivos de mayor magnitud utilizando la misma cantidad de bits que sus equivalentes con signo.

Tipo	Valor mínimo	Valor máximo
<code>u8</code>	0	255
<code>u16</code>	0	65 535
<code>u32</code>	0	4 294 967 295
<code>u64</code>	0	18 446 744 073 709 551 615
<code>u128</code>	0	$2^{128} - 1$

Tabla 2: Rangos numéricos de los tipos enteros sin signo

Estos tipos son ideales para contadores, índices de arrays, y cualquier magnitud que por definición no pueda ser negativa.

```
let edad: u8 = 25;
let habitantes: u32 = 8_000_000;
let bytes_procesados: u64 = 18_446_744_073_709_551_615;
```

Rust

Tipos Dependientes de Arquitectura: `usize` e `isize`

A diferencia de los tipos de tamaño fijo, `usize` e `isize` tienen un tamaño que se adapta automáticamente a la arquitectura del procesador donde se compila el programa. En sistemas de 32 bits, estos tipos equivalen a `u32` e `i32` respectivamente; en sistemas de 64 bits, a `u64` e `i64`.

Tipo	Valor mínimo	Valor máximo
<code>usize</code>	0	$2^{32} - 1$ o $2^{64} - 1$
<code>isize</code>	-2^{31} o -2^{63}	$2^{31} - 1$ o $2^{63} - 1$

Tabla 3: Rangos numéricos de los tipos enteros dependientes de arquitectura

Estos tipos se utilizan principalmente para:

- Indexar colecciones (vectores, arrays, slices)
- Representar tamaños de memoria y longitudes
- Realizar aritmética de punteros

```
let indice: usize = 2;
```

Rust

Esta adaptabilidad permite que el mismo código funcione eficientemente tanto en microcontroladores de 32 bits como en servidores de 64 bits, sin modificaciones.

Inferencia de Tipos Enteros

Cuando no se especifica explícitamente el tipo de un literal entero, Rust aplica la inferencia de tipos y asume `i32` como valor predeterminado. Esta elección se fundamenta en que `i32` ofrece un equilibrio óptimo entre rango numérico y rendimiento en la mayoría de las arquitecturas modernas.

```
let numero = 42; // Tipo inferido: i32
```

Rust

```
let negativo = -100;    // Tipo inferido: i32
let resultado = numero + negativo; // i32
```

Especificación Explícita de Tipos

Anotación de Tipo

La forma más común de especificar el tipo es mediante anotaciones:

```
let byte: u8 = 255;
let contador: u32 = 1_000_000;
let timestamp: i64 = 1_234_567_890;
```

Rust

Sufijos de Tipo en Literales

Rust permite especificar el tipo directamente en el literal numérico mediante sufijos:

```
let a = 100i32;    // i32 explícito
let b = 255u8;     // u8 explícito
let c = 1_000i64;  // i64 explícito
let d = 500usize;  // usize explícito
```

Rust

Esta sintaxis es especialmente útil en expresiones donde la inferencia de tipos podría ser ambigua.

Representación de Literales Enteros

Separadores Visuales

Para mejorar la legibilidad de números grandes, Rust permite el uso del guion bajo (`_`) como separador visual. Este separador no afecta el valor numérico y puede colocarse en cualquier posición:

```
let millones = 1_000_000;
let billion = 1_000_000_000;
let bits = 0b1111_0000_1010_1010;
let hex = 0xdead_beef;
```

Rust

Bases Numéricas

Rust soporta la representación de enteros en cuatro bases numéricas diferentes mediante prefijos específicos:

```
let decimal = 255;           // Base 10 (predeterminada)
let hexadecimal = 0xff;      // Base 16 (prefijo 0x)
let octal = 0o377;           // Base 8 (prefijo 0o)
let binario = 0b1111_1111;   // Base 2 (prefijo 0b)
```

Rust

Adicionalmente, Rust proporciona literales de byte para representar valores ASCII:

```
let byte_a: u8 = b'A';      // Equivale a 65
let byte_newline: u8 = b'\n'; // Equivale a 10
```

Rust

Desbordamiento de Enteros

El desbordamiento ocurre cuando una operación aritmética produce un resultado que excede el rango permitido por el tipo. El comportamiento de Rust ante el desbordamiento depende del perfil de compilación:

Modo debug:

Rust inserta verificaciones automáticas que provocan un pánico en tiempo de ejecución cuando se detecta un desbordamiento, facilitando la detección temprana de errores.

```
let x: u8 = 255;
let y = x + 1; // Pánico: intento de sumar con desbordamiento
```

Rust

Modo release:

Por razones de rendimiento, las verificaciones se eliminan y el desbordamiento produce un comportamiento de **wrapping**, donde el valor “da la vuelta” al rango válido.

```
let x: u8 = 255;
let y = x + 1; // y = 0 (wrapping sin pánico)
```

Rust

Control Explícito del Desbordamiento

Para manejar el desbordamiento de forma predecible independientemente del perfil de compilación, Rust proporciona métodos específicos:

```
let x: u8 = 255;

// Wrapping: siempre da la vuelta
let a = x.wrapping_add(1); // 0

// Checked: devuelve Option<T>
let b = x.checked_add(1); // None

// Saturating: se detiene en el límite
let c = x.saturating_add(1); // 255

// Overflowing: retorna valor y flag booleano
let (d, overflow) = x.overflowing_add(1); // (0, true)
```

Rust

Conversiones entre Tipos Enteros

Rust no realiza conversiones implícitas entre tipos enteros, incluso cuando la conversión sería segura. Todas las conversiones deben ser explícitas utilizando el operador **as**:

```
let a: u8 = 100;
let b: u16 = a as u16; // Conversión segura (widening)
let c: u32 = b as u32;

let x: u32 = 1000;
let y: u8 = x as u8; // Conversión potencialmente peligrosa (narrowing)
// y = 232 (se truncan los bits superiores)
```

Rust

Advertencia: Las conversiones que reducen el tamaño del tipo (**narrowing**) pueden resultar en pérdida de datos si el valor excede el rango del tipo destino. Rust trunca los bits superiores sin advertencia.

Constantes de Rango

Todos los tipos enteros proporcionan constantes asociadas que definen sus límites numéricos:

```
println!("Rango de u8: {} a {}", u8::MIN, u8::MAX);
println!("Rango de i16: {} a {}", i16::MIN, i16::MAX);
println!("Rango de u32: {} a {}", u32::MIN, u32::MAX);
println!("Rango de isize: {} a {}", isize::MIN, isize::MAX);
```

Rust

Operaciones y métodos comunes

```
let x: i32 = 42;

println!("Abs: {}", x.abs());           // valor absoluto
println!("Pow: {}", x.pow(3));          // potencia (42^3)
println!("{}", x.is_positive());        // es positivo?
println!("{}", x.is_negative());        // es negativo?
println!("{}", x.to_string());          // convierte a una cadena de texto
println!("{}", x.signum());
```

Rust

Operador	Descripción	Ejemplo	Resultado
+	Suma	let numero = 15 + 5;	20
-	Resta	let numero = 15 - 5;	10
*	Multiplicación	let numero = 15 * 5;	75
/	División	let numero = 15 / 5;	3
%	Módulo	let numero = 15 % 5;	0

Tabla 4: Operadores aritméticos

Operador	Equivalente a	Ejemplo
<code>+=</code>	<code>x = x + y</code>	<code>x += 2;</code>
<code>-=</code>	<code>x = x - y</code>	<code>x -= 3;</code>
<code>*=</code>	<code>x = x * y</code>	<code>x *= 5;</code>
<code>/=</code>	<code>x = x / y</code>	<code>x /= 2;</code>
<code>%=</code>	<code>x = x % y</code>	<code>x %= 3;</code>

Tabla 5: Operadores de asignación compuesta

Punto Flotante

Los tipos de punto flotante (**floating-point**) permiten representar números con componente fraccionaria. Estos tipos son fundamentales en cálculos de alta precisión.

Rust implementa el estándar IEEE 754 para aritmética de punto flotante, garantizando compatibilidad con otros lenguajes y sistemas, y proporcionando un comportamiento predecible en diferentes plataformas.

Tipos Disponibles

Rust proporciona dos tipos de punto flotante que difieren en su precisión y rango de representación:

Tipo	Precisión	Rango aproximado
f32	6–9 dígitos significativos	$\pm 3.4 \times 10^{38}$
f64	15–17 dígitos significativos	$\pm 1.8 \times 10^{308}$

Tabla 6: Características de los tipos de punto flotante según IEEE 754

Precisión simple f32 : Utiliza 32 bits para almacenar el valor. Es más eficiente en términos de memoria y puede ofrecer ventajas de rendimiento.

Precisión doble f64 : Es el tipo predeterminado en Rust debido a que los procesadores modernos operan con f64 a velocidades comparables a f32 , mientras que proporciona significativamente mayor precisión.

```
let pi_aproximado = 3.14159265359; // f64 predeterminado
let euler: f32 = 2.71828; // f32 explícito
let gravedad: f64 = 9.80665; // f64 explícito
```

Inferencia de Tipos

Cuando no se especifica el tipo, Rust infiere `f64` como predeterminado para literales con punto decimal:

```
let x = 3.14;           // Tipo inferido: f64
let y = 2.5;            // Tipo inferido: f64
```

```
let resultado = x * y;    // f64
```

Para forzar el uso de `f32`, se debe especificar mediante anotación de tipo o sufijo:

```
let a: f32 = 3.14;        // Anotación de tipo
let b = 2.5f32;           // Sufijo de tipo
```

Rust

Limitaciones de Precisión

Advertencia importante: Los números de punto flotante no pueden representar todos los valores decimales con exactitud absoluta. Esta limitación es inherente a la representación binaria utilizada por el estándar IEEE 754 y afecta a todos los lenguajes de programación modernos.

Ejemplo clásico de imprecisión:

```
let resultado = 0.1 + 0.2;
println!("{}", resultado); // Salida: 0.30000000000000004
```

Rust

Valores Especiales

El estándar IEEE 754 define valores especiales para representar condiciones excepcionales:

```
// Infinito positivo y negativo
println!("Infinito: {}", f64::INFINITY);
println!("Infinito negativo: {}", f64::NEG_INFINITY);

// Not a Number (NaN)
println!("NaN: {}", f64::NaN);

// Operaciones que producen valores especiales
let div_cero = 1.0 / 0.0;           // INFINITY
let div_cero_neg = -1.0 / 0.0;      // NEG_INFINITY
let raiz_negativa = (-1.0_f64).sqrt(); // NaN
```

Rust

Verificación de Valores Especiales

Rust proporciona métodos para detectar estos casos:

```
let valor = 1.0 / 0.0;

println!("¿Es infinito?: {}", valor.is_infinite());
println!("¿Es finito?: {}", valor.is_finite());
println!("¿Es NaN?: {}", valor.is_nan());
println!("¿Es signo negativo?: {}", valor.is_sign_negative());
println!("¿Es signo positivo?: {}", valor.is_sign_positive());
```

Rust

Constantes y Límites

Todos los tipos de punto flotante proporcionan constantes útiles:

```
// Límites numéricos
```

Rust

```
println!("f32 min: {}, max: {}", f32::MIN, f32::MAX);  
// -3.4028235e38 3.4028235e38  
println!("f64 min: {}, max: {}", f64::MIN, f64::MAX);  
// -1.7976931348623157e308 1.7976931348623157e308
```

Operaciones Matemáticas

Rust proporciona una amplia colección de métodos matemáticos para tipos de punto flotante:

Redondeo

```
let valor = 3.7; Rust  
  
println!("floor (hacia abajo): {}", valor.floor()); // 3.0  
println!("ceil (hacia arriba): {}", valor.ceil()); // 4.0  
println!("round (más cercano): {}", valor.round()); // 4.0  
println!("trunc (parte entera): {}", valor.trunc()); // 3.0
```

Operaciones con Signo

```
let negativo = -15.8; Rust  
  
println!("Valor absoluto: {}", negativo.abs()); // 15.8  
println!("Signum (+1, 0 o -1): {}", negativo.signum()); // -1.0  
println!("Copiar signo: {}", 10.0_f64.copysign(negativo)); // -10.0
```

Potencias y Raíces

```
let base = 4.0; Rust  
  
println!("Potencia entera: {}", base.powi(3)); // 64.0  
println!("Potencia decimal: {}", base.powf(1.5)); // 8.0  
println!("Raíz cuadrada: {}", base.sqrt()); // 2.0  
println!("Raíz cúbica: {}", 27.0_f64.cbrt()); // 3.0
```

Descomposición de Valores

```
let numero:f32 = 42.75; Rust  
  
println!("Parte entera: {}", numero.trunc()); // 42.0  
println!("Parte fraccionaria: {}", numero.fract()); // 0.75
```

Formato de Salida

Rust proporciona especificadores de formato para controlar la presentación de números de punto flotante:

```
let valor = 123.456789; Rust  
  
// Controlar decimales  
println!("2 decimales: {:.2}", valor); // 123.46  
println!("5 decimales: {:.5}", valor); // 123.45679
```



```
// Notación científica
println!("Científica minúscula: {:e}", valor); // 1.23456789e2
println!("Científica mayúscula: {:E}", valor); // 1.23456789E2

// Ancho y alineación
println!("Ancho 10, 2 dec: {:.10.2}", valor); // "      123.46"
println!("Alineado izq: {:<10.2}", valor);    // "123.46   "
```

Conversiones entre Tipos

Al igual que con los enteros, las conversiones entre tipos de punto flotante deben ser explícitas:

```
let x: f64 = 3.14159265359;

let y: f32 = x as f32;
// Conversión de f64 a f32 pérdida de precisión
// 3.1415927

let a: f32 = 42.5;

let b: f64 = a as f64; // Conversión de f32 a f64 sin pérdida

// Conversión a enteros trunca la parte decimal
let entero: i32 = x as i32; // 3
```

Rust

Booleano

El tipo `bool` es un tipo primitivo fundamental en Rust que representa valores lógicos. Este tipo es esencial para expresar condiciones, realizar comparaciones y controlar el flujo de ejecución de un programa mediante estructuras condicionales y bucles.

Un valor booleano solo puede adoptar dos estados posibles:

- `true` : verdadero
- `false` : falso

```
let es_mayor_edad: bool = true;
let esta_lloviendo: bool = false;
let resultado: bool = 10 > 5; // true
```

Rust

El tipo `bool` ocupa exactamente 1 byte de memoria, aunque conceptualmente solo requiere 1 bit de información.

Operadores Lógicos

Rust proporciona un conjunto de operadores lógicos para combinar, invertir o comparar valores booleanos. Estos operadores son fundamentales para construir expresiones lógicas complejas.

Operador	Nombre	Descripción
!	NOT : Negación	Invierte el valor lógico: <code>true</code> se convierte en <code>false</code> y viceversa
&&	AND : Conjunción	Devuelve <code>true</code> solo si ambos operandos son <code>true</code>
	OR : Disyunción	Devuelve <code>true</code> si al menos uno de los operandos es <code>true</code>
^	XOR : Disyunción exclusiva	Devuelve <code>true</code> solo si exactamente uno de los operandos es <code>true</code>

Tabla 7: Operadores lógicos para el tipo `bool`

NOT

El operador `!` invierte el valor de una expresión booleana. Es un operador unario que se aplica a un único operando.

Expresión	Resultado
<code>!true</code>	<code>false</code>
<code>!false</code>	<code>true</code>

Tabla 8: Tabla de verdad del operador NOT

AND

El operador `&&` evalúa dos expresiones y devuelve `true` únicamente cuando ambas expresiones son verdaderas. Este operador implementa **short-circuit**: si el primer operando es `false`, el segundo no se evalúa.

A	B	A && B
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>false</code>

Tabla 9: Tabla de verdad del operador AND

Ejemplo práctico:

```
let edad = 25;  
let tiene_licencia = true;
```

Rust

```
let puede_conducir = edad >= 18 && tiene_licencia;
println!("¿Puede conducir? {}", puede_conducir); // true
```

OR

El operador `||` devuelve `true` si al menos uno de los operandos es verdadero. También implementa evaluación perezosa: si el primer operando es `true`, el segundo no se evalúa.

A	B	A B
true	true	true
true	false	true
false	true	true
false	false	false

Tabla 10: Tabla de verdad del operador OR

Ejemplo práctico:

```
let es_fin_semana = true;
let es_feriado = false;
let puede_descansar = es_fin_semana || es_feriado;
println!("¿Puede descansar? {}", puede_descansar); // true
```

Rust

XOR

El operador `^` devuelve `true` únicamente cuando los operandos tienen valores diferentes. Es útil para detectar discrepancias o alternar estados.

A	B	A ^ B
true	true	false
true	false	true
false	true	true
false	false	false

Tabla 11: Tabla de verdad del operador XOR

Ejemplo práctico:

```
let estado_anterior = true;
let estado_actual = false;
let hubo_cambio = estado_anterior ^ estado_actual;
println!("¿Hubo cambio? {}", hubo_cambio); // true
```

Rust

Operadores de Comparación

Los operadores de comparación evalúan relaciones entre valores y devuelven un resultado booleano. Son fundamentales para la construcción de expresiones condicionales.

Operador	Nombre	Descripción
<code>==</code>	Igualdad	Verdadero si ambos valores son iguales
<code>!=</code>	Desigualdad	Verdadero si los valores son diferentes
<code><</code>	Menor que	Verdadero si el izquierdo es menor que el derecho
<code>></code>	Mayor que	Verdadero si el izquierdo es mayor que el derecho
<code><=</code>	Menor o igual	Verdadero si el izquierdo es menor o igual al derecho
<code>>=</code>	Mayor o igual	Verdadero si el izquierdo es mayor o igual al derecho

Tabla 12: Operadores de comparación

Conversiones y Métodos

Conversión a Cadena

El tipo `bool` puede convertir valores booleanos a cadenas de texto:

```
let verdadero = true;
let falso = false;

println!("{}", verdadero.to_string()); // "true"
println!("{}", falso.to_string());    // "false"
```

Rust

Conversión a Enteros

Los valores booleanos pueden convertirse a tipos numéricos mediante el operador `as`. Por convención, `false` se convierte en `0` y `true` en `1`:

```
let verdadero = true;
let falso = false;

let valor_v: u8 = verdadero as u8; // 1
let valor_f: u8 = falso as u8;    // 0
```

Rust

Precedencia de Operadores

Cuando se combinan múltiples operadores, es importante comprender su orden de evaluación:

- `!`: Mayor precedencia
- Operadores de comparación (`==`, `!=`, `<`, `>`, `<=`, `>=`)
- `&&`
- `||`

5. `^`: Menor precedencia

```
let resultado1 = !false && true || false;
// Equivale a: ((!false) && true) || false
// = (true && true) || false
// = true || false
// = true

// Con paréntesis explícitos
let resultado2 = (!false && true) || false;
// true
```

Rust

Recomendación: Aunque Rust tiene reglas claras de precedencia, el uso de paréntesis explícitos mejora la legibilidad del código y previene errores sutiles.

Carácter

El tipo `char` en Rust es una representación completa y segura de cualquier carácter Unicode.

Representación de Caracteres

Los literales de tipo `char` se escriben entre comillas simples (`' '`), distinguiéndose así de las cadenas de texto que utilizan comillas dobles (`" "`):

```
let letra: char = 'A';
let minuscula: char = 'ñ';
let numero: char = '7';
let simbolo: char = '©';
let emoji: char = '😄';
let kanji: char = '字';
let arabe: char = 'ع';
```

Rust

Métodos de Inspección

Rust proporciona una amplia colección de métodos para analizar las propiedades de un carácter:

Clasificación de Caracteres

```
let c = 'A';

// Propiedades alfabéticas
println!("¿Alfabético?: {}", c.is_alphabetic()); // true
println!("¿Alfanumérico?: {}", c.is_alphanumeric()); // true
println!("¿Mayúscula?: {}", c.is_uppercase()); // true
println!("¿Minúscula?: {}", c.is_lowercase()); // false

// Propiedades numéricas
let numero = '7';
println!("¿Dígito?: {}", numero.is_numeric()); // true
println!("¿Dígito ASCII?: {}", numero.is_ascii_digit()); // true
```

Rust

```
// Espacios en blanco
let espacio = ' ';
println!("¿Espacio?: {}", espacio.is_whitespace()); // true

// Control y formato
let tab = '\t';
println!("¿Control?: {}", tab.is_control()); // true
```

Secuencias de Escape

Rust soporta varias secuencias de escape para representar caracteres especiales:

Secuencia	Carácter	Descripción
<code>\n</code>	Nueva línea	Line feed (LF)
<code>\r</code>	Retorno de carro	Carriage return (CR)
<code>\t</code>	Tabulador	Tabulador horizontal
<code>\\</code>	Barra invertida	Backslash literal
<code>\'</code>	Comilla simple	Necesaria en literales char
<code>\0</code>	Carácter nulo	Byte cero
<code>\x7F</code>	ASCII hex	Carácter ASCII en hexadecimal (2 dígitos)
<code>\u{1F680}</code>	Unicode	Carácter Unicode (hasta 6 dígitos hex)

Tabla 13: Secuencias de escape para caracteres

```
// Secuencias comunes
let nueva_linea = '\n';
let tab = '\t';
let comilla = '\'';
let backslash = '\\';

// ASCII hexadecimal
let delete = '\x7F'; // Carácter DEL

// Unicode con código
let cohete = '\u{1F680}'; // 🚀
let corazon = '\u{2764}'; // ❤️

println!("Cohete: {}", cohete);
println!("Corazón: {}", corazon);
```

Rust

Compound Types

En Rust, los tipos compuestos son aquellos que combinan varios valores en una sola unidad de datos.

A diferencia de los tipos escalares, los compuestos pueden agrupar o contener múltiples valores de uno o varios tipos.

Rust tiene dos tipos compuestos principales:

- Tuplas “tuple”
- Arreglos “array”

Tuplas

En Rust, una tupla es un tipo compuesto que permite agrupar varios valores dentro de un único contenedor. Su tamaño es fijo: una vez creada, no puede crecer ni reducirse.

Una tupla es útil cuando:

- Quieres empaquetar distintos tipos de datos.
- No necesitas asignar nombre a cada campo.
- Quieres retornar varios valores desde una función.
- Deseas manipular datos agrupados de manera temporal y ligera.

Rust las define como `(valor1, valor2, valor3, ...)`.

Características principales

- **Puede contener valores de tipos distintos.**

`(i32, f64, &str)`

- **El orden importa.**

`(1, "Hola")` es diferente de `("Hola", 1)`.

- **Su tamaño es fijo.**

No se pueden agregar ni remover elementos.

- **Son ligeras y rápidas.**

Las tuplas viven en la stack cuando es posible, por lo que no requieren asignación dinámica.

- **Tipos heterogéneos.**

A diferencia de los arrays, las tuplas pueden contener elementos de diferentes tipos en una misma estructura.

Creación de tuplas

Se definen entre paréntesis:

```
let persona: (&str, i32, bool) = ("Luis", 24, true);
```

Rust

Rust también puede inferir los tipos:

```
let persona = ("Luis", 24, true); // (&str, i32, bool)
```

Rust

Acceso por índice

Cada valor dentro de la tupla se accede usando un índice con punto ‘.’

Los índices empiezan en 0.

```
println!("Nombre: {}", persona.0); // elemento 1: Luis
println!("Edad: {}", persona.1);   // elemento 2: 24
println!("Activa: {}", persona.2); // elemento 3: true
```

Rust

Desestructuración

Desempaqueta la tupla en variables:

```
let persona = ("Luis", 25, true);
let (nombre, edad, activo) = persona;

println!("{}", nombre); // Luis
println!("{}", edad);   // 25
println!("{}", activo); // true
```

Rust

Ignorar valores con ‘_’

```
let persona = ("Luis", 25, true);
let (_, edad, _) = persona;

println!("{}", edad); // 25
```

Rust

Debug en tuplas

Puedes imprimir una tupla con:

```
println!("{:?}", persona); // ("Luis", 24, true)
```

Rust

Siempre que todos sus elementos implementen `Debug`.

Mutabilidad

La tupla completa debe ser mutable para modificar uno de sus campos.

Ejemplo:

```
let mut persona = ("Luis", 25, true);
persona.0 = "Ana";
persona.1 = 30;
persona.2 = false;

println!("{:?}", persona);
```

Rust

Resultado:

```
("Ana", 30, false)
```

Output

Tupla vacía

Rust tiene una tupla especial: la tupla vacía ‘`()`’.

También se llama:

- unit type
- unit value

```
fn saludo() {  
    println!("Hola!");  
}  
  
fn main() {  
    let x = saludo();  
    println!("{:?}", x); // imprime ()  
}
```

Rust

Cuando una función no retorna nada, en realidad retorna Unit type.

Rest pattern

Puedes usar ‘`..`’ para ignorar múltiples elementos intermedios:

```
let tupla = (1, 2, 3, 4, 5);  
let (primero, .., ultimo) = tupla;  
  
println!("primero = {}, ultimo = {}", primero, ultimo);  
// primero = 1, ultimo = 5
```

Rust

```
let tupla = (1, 2, 3, 4, 5);  
let (primero, segundo, ..) = tupla;  
  
println!("{}", {}, {}, primero, segundo); // 1, 2
```

Rust

Tuplas anidadas

Puedes combinar tuplas dentro de otras:

```
let anidada = ((1, 2), (3, 4));  
  
println!("{}", (anidada.0).1); // 2  
println!("{}", (anidada.1).0); // 3
```

Rust

Para mayor claridad, puedes desestructurar:

```
let anidada = ((1, 2), (3, 4));  
let ((a, b), (c, d)) = anidada;  
  
println!("a={}, b={}, c={}, d={}", a, b, c, d);  
// a=1, b=2, c=3, d=4
```

Rust

Tuplas con un solo elemento

Esto confunde a muchos:

```
let x = (5);    // esto NO es una tupla, solo es un i32
let y = (5,);   // esto SÍ es una tupla de un elemento
```

Rust

Las tuplas de un solo elemento deben llevar coma final.

Comparaciones

Las tuplas se pueden comparar si todos sus elementos implementan los traits necesarios:

```
fn main() {
    let t1 = (1, 2);
    let t2 = (1, 3);

    println!("{}", t1 < t2);    // true
    println!("{}", t1 == t2);   // false
}
```

Rust

La comparación se hace elemento por elemento, de izquierda a derecha (orden lexicográfico).

Límite de elementos

Rust implementa automáticamente traits como `Debug`, `Clone`, `PartialEq`, etc., para tuplas de hasta **12 elementos**.

```
// Esto funciona sin problema
let t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
println!("{}", t);
```

Rust

Para tuplas con más de 12 elementos, necesitarás implementar manualmente estos traits si los necesitas:

```
// Esto compila, pero Debug no está implementado automáticamente
let t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
// println!("{}", t); // Error: Debug no implementado
```

Rust

Arrays

Un array en Rust es una colección de tamaño fijo de valores del mismo tipo.

Los arrays de tamaño fijo siempre se almacenan en el stack.

Esto los hace muy rápidos, pero limita su uso a tamaños pequeños o medianos.

Una vez definido, su tamaño no puede cambiar durante la ejecución del programa.

Sintaxis general

```
let nombre: [Tipo; tamaño] = [valor1, valor2, valor3, ...];
```

Rust

Creación

Se definen entre corchetes `[]`.

```
let notas: [i32; 4] = [15, 18, 20, 17];
```

Rust

```
// Índice: 0, 1, 2, 3
```

- `let notas: [i32; 4]` se crea un array con 4 elementos `i32`
- `[15, 18, 20, 17]` se establecen los 4 números separados por `,`.

Debug e impresión

Puedes imprimir arrays fácilmente con `{:?}` o `{:#?}` para formato legible.

```
let datos = [10, 20, 30];  
println!("{:?}", datos); // [10, 20, 30]  
println!("{:#?}", datos); // formato en líneas
```

Rust

Resultado con `{:#?}` :

```
[  
  10,  
  20,  
  30,  
]
```

Rust

Tipos de datos

Los arrays pueden ser de cualquier tipo conocido en tiempo de compilación:

```
let cadenas = ["uno", "dos", "tres"];  
let booleanos = [true, false, true];  
let caracteres = ['a', 'b', 'c'];  
let flotantes = [1.5, 2.7, 3.9];
```

Rust

Si mezclas tipos distintos, Rust no compila:

```
// Error de compilación:  
let mixto = [1, "dos", 3.0];
```

Rust

Acceso por índice

```
let notas = [15, 18, 20, 17];  
  
println!("Primera nota: {}", notas[0]); // 15  
println!("Segunda nota: {}", notas[1]); // 18  
println!("Tercera nota: {}", notas[2]); // 20  
println!("Cuarta nota: {}", notas[3]); // 17
```

Rust

Los índices empiezan en 0, igual que en tuplas.

Acceso seguro

Rust evita errores de índice fuera de rango en tiempo de compilación cuando es posible.

```
let arr = [1, 2, 3];
```

Rust

```
println!("{}", arr[5]); // panic!
```

Acceder fuera del rango genera un panic!.

Para acceso seguro sin panic, usa métodos como `get()` :

```
let arr = [1, 2, 3];

match arr.get(5) {
    Some(valor) => println!("Valor: {}", valor),
    None => println!("Índice fuera de rango"),
}

// Imprime: Índice fuera de rango
```

Rust

Declaración de arrays

Inicialización explícita

```
let valores = [5, 10, 15]; // tipo inferido: [i32; 3]
```

Rust

Inicialización repetida

```
let ceros = [0; 5]; // [0, 0, 0, 0, 0]
```

Rust

Esto crea un array de 5 elementos, todos con valor 0.

Útil para inicializar arrays grandes:

```
let buffer = [0u8; 1024]; // 1024 bytes en cero
```

Rust

Mutabilidad

Para modificar un array, debe ser mutable con `mut` .

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

println!("{:?}", numeros); // Antes: [1, 2, 3, 4]

numeros[2] = 99; // modificamos el valor 3 por 99

println!("{:?}", numeros); // Después: [1, 2, 99, 4]
```

Rust

No puedes cambiar el tamaño, solo los valores:

```
let mut arr = [1, 2, 3];

// arr[3] = 4; // Error: índice fuera de rango
```

Rust

Desestructuración

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

Rust

```
// Desestructuración completa
let [a, b, c, d, e] = numeros;

println!("a={}, b={}, c={}, d={}, e={}", a, b, c, d, e);
// a=1, b=2, c=3, d=4, e=5
```

El número de variables debe coincidir con el tamaño del array.

Desestructuración parcial

Si no te interesan todos los valores, puedes usar `_` para ignorar:

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

let [primero, _, tercero, _] = numeros;

println!("Primero = {}, Tercero = {}", primero, tercero);
// Primero = 10, Tercero = 30
```

Rust

Rest pattern

Puedes tomar los primeros elementos y manejar el resto:

```
let numeros = [100, 200, 300, 400, 500];

// Ignorar el resto
let [x, y, ..] = numeros;
println!("x = {}, y = {}", x, y);

// Capturar el resto en una slice
let [a, b, rest @ ..] = numeros;
println!("a = {}, b = {}, resto = {:?}", a, b, rest);
```

Rust

Resultado:

```
x = 100, y = 200
a = 100, b = 200, resto = [300, 400, 500]
```

Output

También puedes capturar desde el final:

```
let nums = [1, 2, 3, 4, 5];
let [.., penultimo, ultimo] = nums;

println!("{}", {}, penultimo, ultimo); // 4, 5
```

Rust

Comparaciones

Puedes comparar arrays directamente si sus elementos implementan `PartialEq` o `Ord`.

```
let a = [1, 2, 3];
```

Rust

```
let b = [1, 2, 3];
let c = [1, 2, 4];

println!("{}", a == b); // true
println!("{}", a != c); // true
println!("{}", a < c); // true (comparación lexicográfica)
```

La comparación es elemento por elemento, de izquierda a derecha.

Copia y movimiento

Los arrays se copian completamente si contienen tipos que implementan el trait `Copy` como enteros, booleanos o caracteres.

```
let a = [1, 2, 3];
let b = a; // copia completa del array

println!("{}", a); // válido, no se mueve
println!("{}", b); // [1, 2, 3]
```

Esto ocurre porque los arrays de tamaño fijo están en el stack y son baratos de copiar.

Arrays con tipos que no implementan Copy

```
let a = [String::from("Hola"), String::from("Mundo")];
// let b = a; //Error: String no implementa Copy

// Solución: clonar explícitamente
let b = a.clone();
```

Arrays multidimensionales

```
let matriz: [[i32; 3]; 2] = [
    [1, 2, 3],
    [4, 5, 6],
];

println!("Elemento fila 0, col 1: {}", matriz[0][1]); // 2
println!("Elemento fila 1, col 2: {}", matriz[1][2]); // 6
```

Se lee de derecha a izquierda: `[[i32; 3]; 2]` = array de 2 filas, cada fila con 3 elementos `i32`.

Arrays 3D:

```
let cubo: [[[i32; 2]; 2]; 2] = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
];

println!("{}", cubo[1][0][1]); // 6
```

Métodos útiles

```
let arr = [5, 10, 15, 20];

println!("Longitud: {}", arr.len());           // 4
println!("Está vacío? {}", arr.is_empty());    // false
println!("Primer elemento: {:?}", arr.first()); // Some(5)
println!("Último elemento: {:?}", arr.last()); // Some(20)
```

Rust

Resultado:

```
Longitud: 4
Está vacío? false
Primer elemento: Some(5)
Último elemento: Some(20)
```

Output

Métodos adicionales útiles

```
let arr = [1, 2, 3, 4, 5];

// Verificar si contiene un valor
println!("{}", arr.contains(&3)); // true

// Obtener slice
let slice = &arr[1..4];
println!("{:?}", slice); // [2, 3, 4]
```

Rust

Conversión entre tuplas y arrays

De tupla a array

```
let tupla: (u32, u32, u32) = (1, 2, 3);
let array: [u32; 3] = tupla.into();

println!("{:?}", array); // [1, 2, 3]
```

Rust

Esto solo funciona para tuplas con elementos del mismo tipo.

De array a tupla

```
let array = [1, 2, 3];
let tupla = (array[0], array[1], array[2]);

println!("{:?}", tupla); // (1, 2, 3)
```

Rust

Límite práctico de tamaño

Aunque técnicamente no hay límite de tamaño para arrays, hay consideraciones prácticas:

- Arrays muy grandes (>10,000 elementos) pueden causar stack overflow

- Para colecciones grandes o dinámicas, usa `Vec<T>` que vive en el heap

```
// ✓ Bien para arrays pequeños
let pequeno = [0; 100];

// Riesgoso para arrays grandes en el stack
// let enorme = [0; 1_000_000]; // Puede causar stack overflow

// Mejor usar Vec para colecciones grandes
let grande = vec![0; 1_000_000];
```

Rust

Collections

A diferencia de los tipos de datos escalares y compuestos, cuyos tamaños son conocidos en tiempo de compilación y que se almacenan generalmente en el stack, las colecciones de Rust están diseñadas para manejar datos de tamaño dinámico y utilizan memoria asignada en el heap.

Una colección es una estructura de datos capaz de almacenar múltiples valores, pero, a diferencia de los arrays o las tuplas, su contenido puede crecer o reducirse dinámicamente durante la ejecución del programa. Esto permite trabajar con cantidades de datos que no se conocen de antemano, como entradas de usuario, resultados de cálculos, datos provenientes de archivos o redes.

Vectores

Un vector en Rust, representado por el tipo `Vec<T>`, es una colección dinámica que permite almacenar una cantidad variable de valores del mismo tipo en una secuencia contigua de memoria. A diferencia de los arrays, cuyo tamaño es fijo y debe conocerse en tiempo de compilación, los vectores pueden crecer o reducirse durante la ejecución del programa.

Creación de vectores

1. Usando la macro `vec![]`

```
let v = vec![1, 2, 3, 4];
println!("{:?}", v); // [1, 2, 3, 4]
```

Rust

2. Creación explícita con `Vec::new()`

```
let mut v: Vec<i32> = Vec::new();
```

Rust

3. Vector con elementos repetidos

```
let v = vec![0; 5]; // [0, 0, 0, 0, 0]
println!("{:?}", v);
```

Rust

4. Vector a partir de otro vector

```
let v1 = vec![1, 2, 3, 4, 5];
let v2 = v1.clone(); // crea una copia exacta

println!("{:?}", v1); // [1, 2, 3, 4, 5]
println!("{:?}", v2); // [1, 2, 3, 4, 5]
```

Rust

5. Vector a partir de un iterador

```
let v: Vec<i32> = (0..5).collect();
println!("{:?}", v); // [0, 1, 2, 3, 4]
```

Rust

6. Vector con capacidad reservada

```
let mut v = Vec::with_capacity(10);
println!("len: {}, cap: {}", v.len(), v.capacity());
// len: 0, cap: 10
```

Rust

Rust reservará espacio en memoria desde el comienzo, evitando realocaciones posteriores.

7. Vector a partir de un array

```
let arr = [1, 2, 3];
let v1 = arr.to_vec();
let v2 = Vec::from([1, 2, 3]);

println!("{:?}", v1); // [1, 2, 3]
println!("{:?}", v2); // [1, 2, 3]
```

Rust

Acceso a elementos

Cuando accedes a un índice en un vector, puedes hacerlo de dos maneras:

1. Usando índice directo `[]`

```
let v = vec![10, 20, 30];
//      0,  1,  2

println!("Primer elemento: {}", v[0]); // 10
println!("Segundo elemento: {}", v[1]); // 20
```

Rust

Si accedes a un índice fuera de rango, el programa entra en `panic` en tiempo de ejecución:

```
let v = vec![10, 20, 30];
// let x = v[5]; // panic: index out of bounds
```

Rust

2. Usando `get()` (recomendado para acceso seguro)

Con `.get()` es más seguro ya que retorna un `Option`:

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

match v.get(1) {
    Some(valor) => println!("Valor: {}", valor), // Valor: 20
    None => println!("Índice fuera de rango"),
}

println!("{:?}", v.get(5)); // None
```

Rust

3. Primera y última posición

```
let v = vec![10, 20, 30, 40];  
  
println!("{:?}", v.first()); // Some(10)  
println!("{:?}", v.last());  // Some(40)
```

Rust

Ambos retornan `Option<T>`.

Mutabilidad

Los vectores pueden modificarse si se declaran con `mut`:

```
let mut v = vec![10, 20, 30];  
  
v[1] = 50; // cambia 20 => 50  
  
println!("{:?}", v); // [10, 50, 30]
```

Rust

También puedes obtener referencias mutables:

```
let mut v = vec![1, 2, 3];  
  
if let Some(primer) = v.first_mut() {  
    *primer = 100;  
}  
  
println!("{:?}", v); // [100, 2, 3]
```

Rust

Agregar elementos

1. Agregar al final con `push()`

```
let mut v = vec![1, 2];  
  
v.push(3);  
v.push(4);  
  
println!("{:?}", v); // [1, 2, 3, 4]
```

Rust

2. Agregar múltiples elementos con `extend()`

```
let mut v = vec![1, 2, 3];  
  
v.extend([4, 5, 6]);  
  
println!("{:?}", v); // [1, 2, 3, 4, 5, 6]
```

Rust

3. Mover todos los elementos de otro vector con `append()`

```
let mut vec1 = vec![1, 2, 3];
let mut vec2 = vec![4, 5, 6];

vec1.append(&mut vec2);

println!("{:?}", vec1); // [1, 2, 3, 4, 5, 6]
println!("{:?}", vec2); // [] (quedó vacío)
```

Rust

`append()` mueve todos los elementos, dejando el segundo vector vacío.

4. Cambiar tamaño con `resize()`

```
let mut v = vec!["hola"];

v.resize(3, "mundo");

println!("{:?}", v); // ["hola", "mundo", "mundo"]
```

Rust

Si el nuevo tamaño es menor, trunca el vector:

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

v.resize(2, 0);

println!("{:?}", v); // [1, 2]
```

Rust

Insertar elementos

Insertar en una posición específica con `insert()`:

```
let mut v = vec![10, 20, 30, 40, 50];
           //      0,  1,  2,  3,  4

v.insert(2, 25); // inserta 25 en índice 2

println!("{:?}", v); // [10, 20, 25, 30, 40, 50]
```

Rust

△ `insert()` desplaza todos los elementos siguientes, por lo que es una operación $O(n)$.

Eliminar elementos

1. Eliminar por índice con `remove()`

```
let mut v = vec![10, 20, 30, 40];

let eliminado = v.remove(1); // elimina el elemento en índice 1

println!("Eliminado: {}", eliminado); // 20
println!("{:?}", v); // [10, 30, 40]
```

Rust

2. Eliminar el último elemento con `pop()`

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

let ultimo = v.pop(); // retorna Option<T>

println!("{:?}", ultimo); // Some(3)
println!("{:?}", v);      // [1, 2]
```

Rust

Si el vector está vacío, `pop()` retorna `None` :

```
let mut v: Vec<i32> = vec![];

println!("{:?}", v.pop()); // None
```

Rust

3. Eliminar un rango con `drain()`

```
let mut v = vec![10, 20, 30, 40, 50];
           //      0,  1,  2,  3,  4

let drenado: Vec<i32> = v.drain(1..4).collect();

println!("Drenado: {:?}", drenado); // [20, 30, 40]
println!("Vector: {:?}", v);       // [10, 50]
```

Rust

`drain()` retorna un iterador con los elementos eliminados.

4. Vaciar todo el vector con `clear()`

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

v.clear(); // elimina todos los elementos

println!("{:?}", v); // []
println!("Longitud: {}", v.len()); // 0
```

Rust

La capacidad se mantiene:

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

println!("Capacidad antes: {}", v.capacity());

v.clear();

println!("Capacidad después: {}", v.capacity()); // misma capacidad
```

Rust

5. Acortar con `truncate()`

Mantiene los primeros `n` elementos y elimina el resto:

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

v.truncate(2);
```

Rust

```
println!("{:?}", v); // [1, 2]
```

Para vaciar todo:

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

Rust

```
v.truncate(0);
```

```
println!("{:?}", v); // []
```

6. Eliminar con reemplazo `swap_remove()`

El elemento eliminado se reemplaza por el último elemento del vector (más eficiente pero no mantiene el orden):

```
let mut v = vec!["foo", "bar", "baz", "qux"];
```

Rust

```
let eliminado = v.swap_remove(1); // elimina "bar"
```

```
println!("Eliminado: {}", eliminado); // bar
```

```
println!("{:?}", v); // ["foo", "qux", "baz"]
```

Ventaja: $O(1)$ en lugar de $O(n)$.

7. Eliminar elementos que cumplen condición con `retain()`

```
let mut v = vec![1, 2, 3, 4, 5, 6];
```

Rust

```
v.retain(|&x| x % 2 == 0); // mantener solo pares
```

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

8. Eliminar duplicados consecutivos con `dedup()`

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

Rust

```
v.dedup();
```

```
println!("{:?}", v); // [1, 2, 3, 4]
```

Para eliminar todos los duplicados (no solo consecutivos), primero ordena:

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

Rust

```
v.sort();
```

```
v.dedup();
```

```
println!("{:?}", v); // [1, 2, 3]
```

Intercambio y manipulación

1. Intercambiar elementos con `swap()`

```
let mut v = vec!["a", "b", "c", "d", "e"];

v.swap(1, 3); // intercambia índices 1 y 3

println!("{:?}", v); // ["a", "d", "c", "b", "e"]
```

Rust

2. Invertir el vector con `reverse()`

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

v.reverse();

println!("{:?}", v); // [5, 4, 3, 2, 1]
```

Rust

3. Rotar elementos

Rotar a la izquierda:

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

v.rotate_left(2);

println!("{:?}", v); // [3, 4, 5, 1, 2]
```

Rust

Rotar a la derecha:

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

v.rotate_right(2);

println!("{:?}", v); // [4, 5, 1, 2, 3]
```

Rust

4. Dividir el vector con `split_off()`

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

let segunda_parte = v.split_off(3);

println!("Primera parte: {:?}", v); // [1, 2, 3]
println!("Segunda parte: {:?}", segunda_parte); // [4, 5]
```

Rust

Longitud y capacidad

Rust distingue entre:

- `len()` : cantidad actual de elementos
- `capacity()` : espacio reservado en memoria

```
let mut v = Vec::with_capacity(10);
```

Rust

```
println!("len: {}, cap: {}", v.len(), v.capacity());  
// len: 0, cap: 10  
  
v.push(1);  
v.push(2);  
  
println!("len: {}, cap: {}", v.len(), v.capacity());  
// len: 2, cap: 10
```

Cuando la capacidad se llena, el vector duplica automáticamente su espacio en memoria:

```
let mut v = Vec::with_capacity(2);  
  
v.push(1);  
v.push(2);  
println!("cap: {}", v.capacity()); // 2  
  
v.push(3); // se queda sin espacio, realocar  
println!("cap: {}", v.capacity()); // 4 (duplicado)
```

Rust

Métodos relacionados con capacidad:

```
let mut v = vec![1, 2, 3];  
  
// Reservar más capacidad  
v.reserve(10);  
println!("cap: {}", v.capacity()); // al menos 13  
  
// Reducir capacidad al mínimo necesario  
v.shrink_to_fit();  
println!("cap: {}", v.capacity()); // 3  
  
// Reservar exactamente n elementos adicionales  
v.reserve_exact(5);  
println!("cap: {}", v.capacity()); // 8
```

Rust

Copia y movimiento

Los vectores NO implementan `Copy`, solo `Clone`.

Al asignarlos, se mueven, no se copian automáticamente:

```
let v1 = vec![1, 2, 3];  
let v2 = v1; // v1 se mueve a v2  
  
// println!("{:?}", v1); // x Error: valor movido  
println!("{:?}", v2); // ✓ [1, 2, 3]
```

Rust

Para mantener ambos, usa `clone()` :

```
let v1 = vec![1, 2, 3];  
let v2 = v1.clone(); // copia explícita  
  
println!("{:?}", v1); // [1, 2, 3]  
println!("{:?}", v2); // [1, 2, 3]
```

Rust

Comparaciones

Puedes comparar vectores si sus elementos implementan `PartialEq` o `Ord` :

```
let a = vec![1, 2, 3];  
let b = vec![1, 2, 3];  
let c = vec![1, 2, 4];  
  
println!("{}", a == b); // true  
println!("{}", a != c); // true  
println!("{}", a < c); // true (comparación lexicográfica)
```

Rust

La comparación es elemento por elemento, de izquierda a derecha.

Ordenamiento

1. Ordenar con `sort()`

```
let mut v = vec![3, 1, 4, 1, 5];  
  
v.sort();  
  
println!("{:?}", v); // [1, 1, 3, 4, 5]
```

Rust

2. Ordenar de mayor a menor

```
let mut v = vec![3, 1, 4, 1, 5];  
  
v.sort_by(|a, b| b.cmp(a));  
  
println!("{:?}", v); // [5, 4, 3, 1, 1]
```

Rust

3. Ordenar con función personalizada `sort_by_key()`

```
let mut personas = vec![  
    ("Ana", 25),  
    ("Luis", 30),  
    ("Pedro", 20),  
];  
  
personas.sort_by_key(|persona| persona.1); // ordenar por edad
```

Rust


```
println!("{:?}", personas);  
// [("Pedro", 20), ("Ana", 25), ("Luis", 30)]
```

4. Ordenamiento inestable (más rápido) `sort_unstable()`

```
let mut v = vec![3, 1, 4, 1, 5];  
  
v.sort_unstable(); // más rápido pero no preserva orden de elementos iguales  
  
println!("{:?}", v); // [1, 1, 3, 4, 5]
```

Rust

Búsqueda

1. Verificar si contiene un elemento con `contains()`

```
let v = vec![10, 20, 30, 40];  
  
println!("{}", v.contains(&20)); // true  
println!("{}", v.contains(&99)); // false
```

Rust

2. Buscar posición con `iter().position()`

```
let v = vec![10, 20, 30, 40];  
  
if let Some(pos) = v.iter().position(|&x| x == 30) {  
    println!("Encontrado en índice: {}", pos); // 2  
}
```

Rust

3. Búsqueda binaria `binary_search()` (requiere vector ordenado)

```
let v = vec![1, 3, 5, 7, 9];  
  
match v.binary_search(&5) {  
    Ok(pos) => println!("Encontrado en: {}", pos), // 2  
    Err(_) => println!("No encontrado"),  
}
```

Rust

Vectores multidimensionales

Puedes crear vectores dentro de otros vectores (matrices dinámicas):

```
let matriz = vec![  
    vec![1, 2, 3],  
    vec![4, 5, 6],  
    vec![7, 8, 9],  
];  
  
println!("{}", matriz[0][1]); // 2
```

Rust

```
println!("{}", matriz[1][2]); // 6
println!("{}", matriz[2][0]); // 7
```

Crear una matriz de tamaño dinámico:

```
let filas = 3;
let columnas = 4;
let mut matriz = vec![vec![0; columnas]; filas];

matriz[1][2] = 99;

println!("{:?}", matriz);
// [[0, 0, 0, 0], [0, 0, 99, 0], [0, 0, 0, 0]]
```

Rust

Métodos de concatenación

1. Concatenar slices con `concat()`

```
let partes = vec!["Hola", "Mundo"];
let resultado = partes.concat();

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

Rust

2. Unir con separador usando `join()`

```
let palabras = vec!["Hola", "desde", "Rust"];
let frase = palabras.join(" ");

println!("{}", frase); // Hola desde Rust
```

Rust

```
let numeros = vec!["1", "2", "3", "4"];
let resultado = numeros.join("-");

println!("{}", resultado); // 1-2-3-4
```

Rust

Métodos de verificación

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

// Verificar si está vacío
println!("{}", v.is_empty()); // false

// Verificar si comienza con una secuencia
println!("{}", v.starts_with(&[1, 2])); // true

// Verificar si termina con una secuencia
println!("{}", v.ends_with(&[4, 5])); // true
```

Rust

Repetir vector

```
let v = vec![1, 2];
let repetido = v.repeat(3);

println!("{:?}", repetido); // [1, 2, 1, 2, 1, 2]
```

Rust

Conversión entre tipos

De vector a array (tamaño conocido)

```
let v = vec![1, 2, 3, 4];
let arr: [i32; 4] = v.try_into().unwrap();

println!("{:?}", arr); // [1, 2, 3, 4]
```

Rust

De vector a slice

```
let v = vec![1, 2, 3, 4, 5];
let slice: &[i32] = &v[1..4];

println!("{:?}", slice); // [2, 3, 4]
```

Rust

Strings

En Rust, trabajar con texto implica conocer dos tipos principales: `String` y `&str`.

`String` es un tipo dinámico, mutable y con propiedad, que vive en el heap.

`&str` es una referencia inmutable a una secuencia de caracteres UTF-8, generalmente llamada “string slice”.

Características de `String`:

- Tamaño dinámico
- Almacenado en el heap
- Ownership
- Codificación UTF-8 válida garantizada
- Es básicamente un `Vec<u8>` con garantía de UTF-8

Características de `&str`:

- Tamaño fijo
- Inmutable
- Puede vivir en el stack, heap o en el binario
- No tiene propiedad (es una referencia)
- También es UTF-8

Creación de Strings

1. String literal: `type &str`

```
let s = "Hola Mundo"; // tipo: &str
println!("{}", s);
```

Rust

Los string literals están en el binario y viven durante todo el programa.

2. Crear `String` desde un literal

```
let s1 = String::from("Hola");  
let s2 = "Hola".to_string();  
let s3 = "Hola".to_owned();  
  
println!("{}", {}, {}, {}, s1, s2, s3);
```

Rust

Todas estas formas crean un `String` en el heap.

3. `String` vacío

```
let mut s = String::new();  
println!("Vacío: '{}'", s); // ''  
println!("Longitud: {}", s.len()); // 0
```

Rust

4. `String` con capacidad reservada

```
let mut s = String::with_capacity(10);  
println!("len: {}, cap: {}", s.len(), s.capacity());  
// len: 0, cap: 10
```

Rust

5. `String` desde bytes

```
let bytes = vec![72, 111, 108, 97]; // "Hola" en UTF-8  
let s = String::from_utf8(bytes).unwrap();  
  
println!("{}", s); // Hola
```

Rust

6. `String` repetido

```
let s = "Rust".repeat(3);  
println!("{}", s); // RustRustRust
```

Rust

Diferencia entre `String` y `&str`

```
fn main() {  
    let s1: String = String::from("Hola"); // owned, heap  
    let s2: &str = "Mundo";                // borrowed, puede estar en binario  
  
    println!("{}", {}, s1, s2);  
}
```

Rust

Conversión entre `String` y `&str`:

```
// De String a &str (fácil, solo tomar prestado)  
let s = String::from("Hola");  
let slice: &str = &s;  
let slice2: &str = s.as_str();  
  
// De &str a String (requiere asignación)
```

Rust

```
let literal = "Hola";
let owned = literal.to_string();
let owned2 = String::from(literal);
```

Longitud y capacidad

```
let mut s = String::with_capacity(10);

println!("len: {}, cap: {}", s.len(), s.capacity());
// len: 0, cap: 10

s.push_str("Hola");

println!("len: {}, cap: {}", s.len(), s.capacity());
// len: 4, cap: 10
```

Rust

`len()` retorna el número de bytes, no el número de caracteres:

```
let s = String::from("Hola");
println!("Bytes: {}", s.len()); // 4

let s = String::from("Hölä");
println!("Bytes: {}", s.len()); // 6 (ö y ä usan 2 bytes cada uno)
```

Rust

Para contar caracteres:

```
let s = "Hölä";
println!("Caracteres: {}", s.chars().count()); // 4
```

Rust

Agregar contenido

1. Agregar un string con `push_str()`

```
let mut s = String::from("Hola");

s.push_str(" Mundo");

println!("{}", s); // Hola Mundo
```

Rust

2. Agregar un carácter con `push()`

```
let mut s = String::from("Hol");

s.push('a');
s.push('!');

println!("{}", s); // Hola!
```

Rust

3. Concatenar con `+`

```
let s1 = String::from("Hola");
let s2 = String::from(" Mundo");

let s3 = s1 + &s2; // s1 se mueve, s2 se presta

// println!("{}", s1); // Error: s1 fue movido
println!("{}", s2);    // s2 sigue disponible
println!("{}", s3);    // Hola Mundo
```

Rust

El operador `+` mueve el primer operando y toma prestado el segundo.

4. Concatenar múltiples strings con `format!`

```
let s1 = String::from("Hola");
let s2 = String::from("Mundo");
let s3 = String::from("Rust");

let resultado = format!("{}", s1, s2, s3);

println!("{}", resultado); // Hola Mundo Rust

// Todos siguen disponibles
println!("{}", s1, s2, s3);
```

Rust

`format!` no mueve ningún valor, solo toma prestado.

5. Insertar en una posición específica

```
let mut s = String::from("Hola Rust");

s.insert(5, 'a'); // inserta en índice de byte 5

println!("{}", s); // Holaa Rust
```

Rust

```
let mut s = String::from("Hola");

s.insert_str(4, " Mundo");

println!("{}", s); // Hola Mundo
```

Rust

Los índices son posiciones de bytes, no de caracteres.

Acceso a caracteres

No puedes acceder directamente por índice:

```
let s = String::from("Hola");
// let c = s[0]; // Error: String no soporta indexación
```

Rust

Usa `chars()` para acceder:

```
let s = String::from("Hola");

// Obtener el primer carácter
if let Some(primer) = s.chars().nth(0) {
    println!("Primer carácter: {}", primero); // H
}

// Recorrer todos los caracteres
for c in s.chars() {
    println!("{}", c);
}
```

Rust

Acceder a bytes (no recomendado para texto)

```
let s = String::from("Hola");

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

// 72, 111, 108, 97
```

Rust

Slicing de strings

Puedes obtener slices usando rangos de bytes:

```
let s = String::from("Hola Mundo");

let hola = &s[0..4]; // "Hola"
let mundo = &s[5..10]; // "Mundo"

println!("{}", hola);
println!("{}", mundo);
```

Rust

Cuidado con caracteres multi-byte:

```
let s = "Hölä";

// let slice = &s[0..2]; // Panic: corta en medio de 'ö'
let slice = &s[0..3]; // "Hö" (ö usa 2 bytes)

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

Rust

Para slicing seguro, verifica límites de caracteres:

```
let s = "Hölä Mundo";

if s.is_char_boundary(3) {
    println!("{}", &s[0..3]); // Hö
}
```

Rust

Métodos de verificación

```
let s = String::from("Hola Mundo");

// Verificar si está vacío
println!("{}", s.is_empty()); // false

// Verificar si empieza con...
println!("{}", s.starts_with("Hola")); // true

// Verificar si termina con...
println!("{}", s.ends_with("Mundo")); // true

// Verificar si contiene...
println!("{}", s.contains("la")); // true

// Comparar (case insensitive)
let s1 = "HOLA";
let s2 = "hola";
println!("{}", s1.eq_ignore_ascii_case(s2)); // true
```

Rust

Búsqueda

1. Buscar primera ocurrencia con `find()`

```
let s = "Hola Mundo Hola";

if let Some(pos) = s.find("Mundo") {
    println!("Encontrado en posición: {}", pos); // 5
}
```

Rust

2. Buscar última ocurrencia con `rfind()`

```
let s = "Hola Mundo Hola";

if let Some(pos) = s.rfind("Hola") {
    println!("Última ocurrencia en: {}", pos); // 11
}
```

Rust

3. Buscar con patrón

```
let s = "Rust 2025";

if let Some(pos) = s.find(char::is_numeric) {
    println!("Primer dígito en posición: {}", pos); // 5
}
```

Rust

4. Contar ocurrencias


```
let s = "Hola Mundo Hola Rust";
let count = s.matches("Hola").count();

println!("'Hola' aparece {} veces", count); // 2
```

Rust

Dividir strings

1. Dividir por un delimitador con `split()`

```
let s = "Rust,Python,JavaScript";

for parte in s.split(',') {
    println!("{}", parte);
}
// Rust
// Python
// JavaScript
```

Rust

Recoger en un vector:

```
let s = "uno,dos,tres";
let partes: Vec<&str> = s.split(',').collect();

println!("{:?}", partes); // ["uno", "dos", "tres"]
```

Rust

2. Dividir por espacios en blanco

```
let s = "Hola Mundo Rust";

let palabras: Vec<&str> = s.split_whitespace().collect();

println!("{:?}", palabras); // ["Hola", "Mundo", "Rust"]
```

Rust

3. Dividir en líneas

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

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

Rust

4. Dividir en n partes con `splitn()`

```
let s = "uno:dos:tres:cuatro";

let partes: Vec<&str> = s.splitn(2, ':').collect();

println!("{:?}", partes); // ["uno", "dos:tres:cuatro"]
```

Rust

5. Dividir desde el final con `rsplit()`

```
let s = "uno,dos,tres";
```

Rust

```
for parte in s.rsplit(',') {  
    println!("{}", parte);  
}  
// tres  
// dos  
// uno
```

Reemplazar contenido

1. Reemplazar todas las ocurrencias con `replace()`

```
let s = "Hola Mundo Hola";
```

Rust

```
let nuevo = s.replace("Hola", "Adiós");  
  
println!("{}", nuevo); // Adiós Mundo Adiós
```

2. Reemplazar n ocurrencias con `replacen()`

```
let s = "Hola Hola Hola";
```

Rust

```
let nuevo = s.replacen("Hola", "Hey", 2);  
  
println!("{}", nuevo); // Hey Hey Hola
```

3. Reemplazar un rango

```
let mut s = String::from("Hola Mundo");
```

Rust

```
s.replace_range(5..10, "Rust");  
  
println!("{}", s); // Hola Rust
```

△ `replace_range()` modifica el String original (requiere `mut`).

Eliminar contenido

1. Eliminar el último carácter con `pop()`

```
let mut s = String::from("Hola!");
```

Rust

```
let ultimo = s.pop();  
  
println!("{:?}", ultimo); // Some('!')  
println!("{}", s); // Hola
```

2. Eliminar un carácter en posición específica con `remove()`

```
let mut s = String::from("Hola");

let eliminado = s.remove(1);

println!("Eliminado: {}", eliminado); // o
println!("String: {}", s);           // Hla
```

Rust

△ Índice debe ser un límite de carácter válido.

3. Eliminar un rango con `drain()`

```
let mut s = String::from("Hola Mundo");

let drenado: String = s.drain(5..10).collect();

println!("Drenado: {}", drenado); // Mundo
println!("String: {}", s);       // Hola
```

Rust

4. Eliminar caracteres del final con `truncate()`

```
let mut s = String::from("Hola Mundo");

s.truncate(4);

println!("{}", s); // Hola
```

Rust

5. Vaciar completamente con `clear()`

```
let mut s = String::from("Hola Mundo");

s.clear();

println!("Vacío: '{}'", s); // ''
println!("Longitud: {}", s.len()); // 0
```

Rust

Transformaciones

1. Convertir a mayúsculas

```
let s = "hola mundo";

let mayus = s.to_uppercase();

println!("{}", mayus); // HOLA MUNDO
```

Rust

2. Convertir a minúsculas

```
let s = "HOLA MUNDO";

let minus = s.to_lowercase();
```

Rust

```
println!("{}", minus); // hola mundo
```

3. Eliminar espacios al inicio y final con `trim()`

```
let s = "  Hola Mundo  ";  
  
let limpio = s.trim();  
  
println!("{}", limpio); // 'Hola Mundo'
```

Rust

4. Eliminar espacios solo al inicio con `trim_start()`

```
let s = "  Hola Mundo  ";  
  
let limpio = s.trim_start();  
  
println!("{}", limpio); // 'Hola Mundo  '
```

Rust

5. Eliminar espacios solo al final con `trim_end()`

```
let s = "  Hola Mundo  ";  
  
let limpio = s.trim_end();  
  
println!("{}", limpio); // '  Hola Mundo'
```

Rust

6. Eliminar caracteres específicos con `trim_matches()`

```
let s = "###Hola###";  
  
let limpio = s.trim_matches('#');  
  
println!("{}", limpio); // Hola
```

Rust

```
let s = "123Hola456";  
  
let limpio = s.trim_matches(char::is_numeric);  
  
println!("{}", limpio); // Hola
```

Rust

Parseo de strings

1. Convertir string a número

```
let s = "42";  
  
let num: i32 = s.parse().unwrap();
```

Rust

```
println!("{}", num + 10); // 52
```

Con manejo de errores:

```
let s = "42";

match s.parse::<i32>() {
    Ok(num) => println!("Número: {}", num),
    Err(e) => println!("Error: {}", e),
}
```

Rust

2. Convertir número a string

```
let num = 42;

let s1 = num.to_string();
let s2 = format!("{}", num);

println!("{}", s1, s2); // 42, 42
```

Rust

Repetir strings

```
let s = "Rust";

let repetido = s.repeat(3);

println!("{}", repetido); // RustRustRust
```

Rust

```
let linea = "-".repeat(50);
println!("{}", linea);
// -----
```

Rust

Comparaciones

```
let s1 = "Hola";
let s2 = "Hola";
let s3 = "Mundo";

println!("{}", s1 == s2); // true
println!("{}", s1 != s3); // true
println!("{}", s1 < s3); // true (orden lexicográfico)
```

Rust

Comparación case-insensitive:

```
let s1 = "HOLA";
let s2 = "hola";

println!("{}", s1.eq_ignore_ascii_case(s2)); // true
```

Rust

Concatenar múltiples strings

1. Usando `format!` (recomendado)

```
let nombre = "Ana";  
let edad = 25;  
  
let mensaje = format!("Me llamo {} y tengo {} años", nombre, edad);  
  
println!("{}", mensaje);
```

Rust

2. Usando el método `join()` para slices

```
let palabras = vec!["Hola", "desde", "Rust"];  
  
let frase = palabras.join(" ");  
  
println!("{}", frase); // Hola desde Rust
```

Rust

3. Concatenar con `+` múltiples veces (no recomendado)

```
let s1 = String::from("Hola");  
let s2 = String::from(" Mundo");  
let s3 = String::from(" Rust");  
  
let resultado = s1 + &s2 + &s3;  
  
println!("{}", resultado); // Hola Mundo Rust
```

Rust

Capacidad y optimización

```
let mut s = String::with_capacity(10);  
  
println!("cap: {}", s.capacity()); // 10  
  
s.push_str("Hola");  
  
println!("len: {}, cap: {}", s.len(), s.capacity());  
// len: 4, cap: 10
```

Rust

Reservar más capacidad:

```
let mut s = String::from("Hola");  
  
s.reserve(10); // reserva al menos 10 bytes adicionales  
  
println!("cap: {}", s.capacity());
```

Rust

Reducir capacidad al mínimo:

```
let mut s = String::with_capacity(100);

s.push_str("Hola");

s.shrink_to_fit();

println!("cap después: {}", s.capacity()); // ≈4
```

Rust

Reservar exactamente:

```
let mut s = String::new();

s.reserve_exact(10);

println!("cap: {}", s.capacity()); // exactamente 10
```

Rust

Validación UTF-8

Rust garantiza que todo `String` es UTF-8 válido:

```
let bytes = vec![72, 111, 108, 97]; // "Hola"

let s = String::from_utf8(bytes).unwrap();

println!("{}", s); // Hola
```

Rust

Con bytes inválidos:

```
let bytes = vec![0, 159, 146, 150]; // UTF-8 inválido

match String::from_utf8(bytes) {
    Ok(s) => println!("{}", s),
    Err(e) => println!("Error: {}", e),
}
```

Rust

Para strings con UTF-8 inválido, usa `from_utf8_lossy()` :

```
let bytes = vec![72, 111, 255, 108, 97];

let s = String::from_utf8_lossy(&bytes);

println!("{}", s); // Ho la (  reemplaza bytes inválidos)
```

Rust

Strings multilinea

```
let poema = "
    Roses are red,
    Violets are blue,
    Rust is awesome,
```

Rust

```
    And so are you!
";

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

Sin espacios al inicio:

```
let texto = "\
Primera línea
Segunda línea
Tercera línea
";

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

Rust

Secuencias de escape

```
let s = "Primera línea\nSegunda línea";
println!("{}", s);

let s = "Tab\taqui";
println!("{}", s);

let s = "Comillas: \"Hola\"";
println!("{}", s);

let s = "Backslash: \\";
println!("{}", s);

// Unicode
let s = "Corazón: \u{2764}";
println!("{}", s); // Corazón: ♥
```

Rust

Raw strings

Para strings literales sin procesar secuencias de escape:

```
let s = r"C:\Users\nombre\Documents";
println!("{}", s); // C:\Users\nombre\Documents
```

Rust

Con comillas:

```
let s = r#"Él dijo: "Hola Mundo" "#;
println!("{}", s);
```

Rust

Con múltiples #:

```
let s = r##"Este string tiene "comillas" y #hashtags"##;
println!("{}", s);
```

Rust

Copia y movimiento

`String` NO implementa `Copy`, solo `Clone`:

```
let s1 = String::from("Hola");  
let s2 = s1; // s1 se mueve  
  
// println!("{}", s1); // x Error: valor movido  
println!("{}", s2); // ✓ Hola
```

Rust

Para copiar, usa `clone()`:

```
let s1 = String::from("Hola");  
let s2 = s1.clone();  
  
println!("{}", s1); // Hola  
println!("{}", s2); // Hola
```

Rust

`&str` sí es `Copy`:

```
let s1 = "Hola";  
let s2 = s1;  
  
println!("{}", s1); // ✓ Hola  
println!("{}", s2); // ✓ Hola
```

Rust

Cuándo usar `String` vs `&str`

Usa `String` cuando:

- Necesitas modificar el texto
- Necesitas propiedad del texto
- El tamaño cambiará dinámicamente
- Construyes strings en tiempo de ejecución

Usa `&str` cuando:

- El texto es de solo lectura
- Quieres prestar texto sin tomar propiedad
- Trabajas con string literals
- Pasas texto a funciones sin modificarlo

Ejemplo en funciones:

```
// Mejor: acepta &str (más flexible)  
fn imprimir(texto: &str) {  
    println!("{}", texto);  
}  
  
fn main() {  
    let s1 = String::from("Hola");  
    let s2 = "Mundo";
```

Rust

```
imprimir(&s1); // ✓ String → &str
imprimir(s2);  // ✓ &str → &str
}
```

HashMap

En Rust, `HashMap<K, V>` es una colección que almacena pares clave-valor, donde cada clave es única y está asociada con un valor.

Es una estructura de datos fundamental para búsquedas rápidas por clave.

`HashMap` vive en el heap y debe importarse desde la biblioteca estándar:

```
use std::collections::HashMap;
```

Rust

Características principales:

- Almacena pares clave-valor `(K, V)`
- Cada clave es única (no hay claves duplicadas)
- Acceso rápido a valores por clave: $O(1)$ en promedio
- No mantiene orden de inserción
- Las claves deben implementar `Eq` y `Hash`
- Tamaño dinámico (puede crecer)
- Vive en el heap

Creación de HashMap

1. HashMap vacío con `new()`

```
use std::collections::HashMap;
```

Rust

```
let mut mapa: HashMap<String, i32> = HashMap::new();
```

```
println!("{:?}", mapa); // {}
```

Siempre necesitas `mut` para modificar un `HashMap`.

2. Usando `from()` con array de tuplas

```
use std::collections::HashMap;
```

Rust

```
let mapa = HashMap::from([
    ("Rust", 2015),
    ("Python", 1991),
    ("JavaScript", 1995),
]);
```

```
println!("{:?}", mapa);
```

3. Con capacidad reservada

```
use std::collections::HashMap;

let mut mapa: HashMap<String, i32> = HashMap::with_capacity(10);

println!("Capacidad: {}", mapa.capacity()); // al menos 10
```

Rust

4. Construir desde vectores

```
use std::collections::HashMap;

let claves = vec!["uno", "dos", "tres"];
let valores = vec![1, 2, 3];

let mut mapa = HashMap::new();

for (k, v) in claves.into_iter().zip(valores.into_iter()) {
    mapa.insert(k, v);
}

println!("{:?}", mapa);
```

Rust

Insertar elementos

1. Insertar con `insert()`

```
use std::collections::HashMap;

let mut capitales = HashMap::new();

capitales.insert("Perú", "Lima");
capitales.insert("Chile", "Santiago");
capitales.insert("Argentina", "Buenos Aires");

println!("{:?}", capitales);
```

Rust

Si la clave ya existe, `insert()` reemplaza el valor anterior:

```
use std::collections::HashMap;

let mut puntos = HashMap::new();

puntos.insert("Ana", 10);
println!("{:?}", puntos); // {"Ana": 10}

puntos.insert("Ana", 20); // reemplaza 10 con 20
println!("{:?}", puntos); // {"Ana": 20}
```

Rust

`insert()` retorna el valor anterior si existía:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let anterior = mapa.insert("clave", 1);
println!("{:?}", anterior); // None

let anterior = mapa.insert("clave", 2);
println!("{:?}", anterior); // Some(1)
```

Rust

Acceso a valores

1. Acceso con `get()` (recomendado)

```
use std::collections::HashMap;

let mut puntos = HashMap::new();
puntos.insert("Ana", 10);
puntos.insert("Luis", 20);

match puntos.get("Ana") {
    Some(valor) => println!("Ana tiene {} puntos", valor),
    None => println!("Ana no encontrada"),
}

// Ana tiene 10 puntos
```

Rust

`get()` retorna `Option<&V>` :

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("clave", 42);

let valor = mapa.get("clave");
println!("{:?}", valor); // Some(42)

let valor = mapa.get("no_existe");
println!("{:?}", valor); // None
```

Rust

2. Acceso con índice `[]` (puede causar panic)

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("clave", 42);

let valor = mapa["clave"];
println!("{}", valor); // 42
```

Rust

```
// let x = mapa["no_existe"]; // x panic: no entry found
```

△ Usar `[]` genera panic si la clave no existe. Prefiere `get()`.

3. Obtener valor o default con `get_or_insert()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let valor = mapa.entry("clave").or_insert(42);
println!("{}", valor); // 42

let valor = mapa.entry("clave").or_insert(100);
println!("{}", valor); // 42 (ya existía)
```

Rust

Verificar existencia

1. Verificar si existe una clave con `contains_key()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("Rust", 2015);

if mapa.contains_key("Rust") {
    println!("Rust existe en el mapa");
}

if !mapa.contains_key("Python") {
    println!("Python no existe en el mapa");
}
```

Rust

2. Verificar si está vacío con `is_empty()`

```
use std::collections::HashMap;

let mut mapa: HashMap<String, i32> = HashMap::new();

println!("{}", mapa.is_empty()); // true

mapa.insert(String::from("clave"), 1);

println!("{}", mapa.is_empty()); // false
```

Rust

Actualizar valores

1. Actualizar directamente con `insert()`

```
use std::collections::HashMap;

let mut puntos = HashMap::new();

puntos.insert("Ana", 10);
puntos.insert("Ana", 20); // actualiza

println!("{:?}", puntos); // {"Ana": 20}
```

Rust

2. Actualizar solo si existe con `get_mut()`

```
use std::collections::HashMap;

let mut puntos = HashMap::new();
puntos.insert("Ana", 10);

if let Some(valor) = puntos.get_mut("Ana") {
    *valor += 5; // incrementa en 5
}

println!("{:?}", puntos); // {"Ana": 15}
```

Rust

3. Insertar solo si no existe con `entry().or_insert()`

```
use std::collections::HashMap;

let mut puntos = HashMap::new();

puntos.entry("Ana").or_insert(10);
println!("{:?}", puntos); // {"Ana": 10}

puntos.entry("Ana").or_insert(20); // no hace nada, ya existe
println!("{:?}", puntos); // {"Ana": 10}
```

Rust

4. Actualizar basado en el valor anterior

```
use std::collections::HashMap;

let mut contador = HashMap::new();

for palabra in vec!["hola", "mundo", "hola", "rust"] {
    let count = contador.entry(palabra).or_insert(0);
    *count += 1;
}

println!("{:?}", contador);
```

Rust

```
// {"hola": 2, "mundo": 1, "rust": 1}
```

5. Insertar con función si no existe con `or_insert_with()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.entry("clave").or_insert_with(|| {
    println!("Calculando valor...");
    42
});

println!("{:?}", mapa); // {"clave": 42}
```

Rust

La función solo se ejecuta si la clave no existe.

Eliminar elementos

1. Eliminar con `remove()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("Rust", 2015);
mapa.insert("Python", 1991);

let eliminado = mapa.remove("Rust");
println!("{:?}", eliminado); // Some(2015)

let no_existe = mapa.remove("Java");
println!("{:?}", no_existe); // None

println!("{:?}", mapa); // {"Python": 1991}
```

Rust

2. Eliminar y obtener par clave-valor con `remove_entry()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("Rust", 2015);

let eliminado = mapa.remove_entry("Rust");
println!("{:?}", eliminado); // Some(("Rust", 2015))
```

Rust

3. Vaciar completamente con `clear()`

```
use std::collections::HashMap;
```

Rust

```
let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);

println!("Antes: {:?}", mapa); // {"a": 1, "b": 2}

mapa.clear();

println!("Después: {:?}", mapa); // {}
println!("Longitud: {}", mapa.len()); // 0
```

4. Mantener solo elementos que cumplen condición con `retain()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);
mapa.insert("c", 3);
mapa.insert("d", 4);

mapa.retain(|_clave, valor| *valor % 2 == 0);

println!("{:?}", mapa); // {"b": 2, "d": 4}
```

Rust

Longitud y capacidad

```
use std::collections::HashMap;

let mut mapa = HashMap::with_capacity(10);

println!("len: {}, cap: {}", mapa.len(), mapa.capacity());
// len: 0, cap: al menos 10

mapa.insert("a", 1);
mapa.insert("b", 2);

println!("len: {}, cap: {}", mapa.len(), mapa.capacity());
// len: 2, cap: al menos 10
```

Rust

Reservar capacidad:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.reserve(100); // reserva espacio para al menos 100 elementos
```

Rust


```
println!("Capacidad: {}", mapa.capacity());
```

Reducir capacidad al mínimo:

```
use std::collections::HashMap;

let mut mapa = HashMap::with_capacity(100);
mapa.insert("a", 1);

mapa.shrink_to_fit();

println!("Capacidad reducida: {}", mapa.capacity());
```

Rust

Recorrer HashMap

1. Recorrer claves y valores

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("Rust", 2015);
mapa.insert("Python", 1991);
mapa.insert("JavaScript", 1995);

for (clave, valor) in &mapa {
    println!("{}", clave, valor);
}
```

Rust

△ El orden NO está garantizado y puede cambiar.

2. Recorrer solo claves

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);
mapa.insert("c", 3);

for clave in mapa.keys() {
    println!("{}", clave);
}
```

Rust

3. Recorrer solo valores

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
```

Rust

```
mapa.insert("b", 2);
mapa.insert("c", 3);

for valor in mapa.values() {
    println!("{}", valor);
}
```

4. Recorrer valores mutables

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);

for valor in mapa.values_mut() {
    *valor *= 2;
}

println!("{:?}", mapa); // {"a": 2, "b": 4}
```

Rust

5. Consumir el HashMap

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);

for (clave, valor) in mapa { // sin &, consume el mapa
    println!("{: {}}", clave, valor);
}

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

Rust

Entry API

La Entry API es poderosa para manipular entradas de forma eficiente.

1. Obtener o insertar con `or_insert()`

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.entry("clave").or_insert(42);
mapa.entry("clave").or_insert(100); // no hace nada

println!("{:?}", mapa); // {"clave": 42}
```

Rust

2. Modificar entrada existente o insertar

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let valor = mapa.entry("contador").or_insert(0);
*valor += 1;

let valor = mapa.entry("contador").or_insert(0);
*valor += 1;

println!("{:?}", mapa); // {"contador": 2}
```

Rust

3. Usar `and_modify()` antes de insertar

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 10);

mapa.entry("a")
    .and_modify(|v| *v += 5)
    .or_insert(1);

mapa.entry("b")
    .and_modify(|v| *v += 5)
    .or_insert(1);

println!("{:?}", mapa); // {"a": 15, "b": 1}
```

Rust

4. Insertar con función solo si no existe

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.entry("clave").or_insert_with(|| {
    println!("Calculando...");
    42
});

mapa.entry("clave").or_insert_with(|| {
    println!("No se ejecuta");
    100
});

println!("{:?}", mapa); // {"clave": 42}
```

Rust

```
// Solo imprime "Calculando..." una vez
```

Combinar HashMaps

1. Extender un HashMap con otro

```
use std::collections::HashMap;

let mut mapa1 = HashMap::new();
mapa1.insert("a", 1);
mapa1.insert("b", 2);

let mut mapa2 = HashMap::new();
mapa2.insert("c", 3);
mapa2.insert("d", 4);

mapa1.extend(mapa2);

println!("{:?}", mapa1); // {"a": 1, "b": 2, "c": 3, "d": 4}
```

Si hay claves duplicadas, los valores del segundo HashMap reemplazan los del primero:

```
use std::collections::HashMap;

let mut mapa1 = HashMap::new();
mapa1.insert("a", 1);

let mut mapa2 = HashMap::new();
mapa2.insert("a", 100);
mapa2.insert("b", 2);

mapa1.extend(mapa2);

println!("{:?}", mapa1); // {"a": 100, "b": 2}
```

Tipos de claves

Las claves deben implementar `Eq` y `Hash`.

Tipos comunes que funcionan:

```
use std::collections::HashMap;

// Números
let mut mapa1: HashMap<i32, &str> = HashMap::new();
mapa1.insert(1, "uno");

// Strings
let mut mapa2: HashMap<String, i32> = HashMap::new();
```

```

mapa2.insert(String::from("clave"), 42);

// &str
let mut mapa3: HashMap<&str, i32> = HashMap::new();
mapa3.insert("clave", 42);

// Tuplas
let mut mapa4: HashMap<(i32, i32), &str> = HashMap::new();
mapa4.insert((0, 0), "origen");

// Chars
let mut mapa5: HashMap<char, i32> = HashMap::new();
mapa5.insert('a', 1);

```

✗ NO puedes usar tipos que no implementan `Eq` y `Hash` :

```

use std::collections::HashMap;

// x f64 no implementa Eq (debido a NaN)
// let mut mapa: HashMap<f64, i32> = HashMap::new();

// x Vec<T> implementa Hash pero no es recomendado
// let mut mapa: HashMap<Vec<i32>, i32> = HashMap::new();

```

Rust

Valores por defecto

Patrón común: contador de palabras

```

use std::collections::HashMap;

let texto = "hola mundo hola rust mundo hola";

let mut contador = HashMap::new();

for palabra in texto.split_whitespace() {
    let count = contador.entry(palabra).or_insert(0);
    *count += 1;
}

println!("{:?}", contador);
// {"hola": 3, "mundo": 2, "rust": 1}

```

Rust

Patrón: agrupar elementos

```

use std::collections::HashMap;

let datos = vec![("Ana", 25), ("Luis", 30), ("Ana", 28)];

```

Rust

```
let mut grupos: HashMap<&str, Vec<i32>> = HashMap::new();

for (nombre, edad) in datos {
    grupos.entry(nombre).or_insert(Vec::new()).push(edad);
}

println!("{:?}", grupos);
// {"Ana": [25, 28], "Luis": [30]}
```

Propiedad (Ownership)

Tipos que implementan Copy:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let clave = 1;
let valor = 10;

mapa.insert(clave, valor);

println!("clave: {}, valor: {}", clave, valor); // ✓ siguen disponibles
```

Rust

Tipos que NO implementan Copy (como String):

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let clave = String::from("nombre");
let valor = String::from("Ana");

mapa.insert(clave, valor);

// println!("{}", clave); // x Error: clave fue movida
// println!("{}", valor); // x Error: valor fue movido
```

Rust

Para mantener las variables originales, clona o usa referencias:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let clave = String::from("nombre");
let valor = String::from("Ana");

mapa.insert(clave.clone(), valor.clone());
```

Rust

```
println!("clave: {}, valor: {}", clave, valor); // ✓ funcionan
```

Usando referencias como claves:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();

let clave = String::from("nombre");
let valor = String::from("Ana");

mapa.insert(&clave, &valor);

println!("clave: {}, valor: {}", clave, valor); // ✓ funcionan
```

Rust

△ Las referencias deben vivir tanto como el HashMap.

Comparar HashMaps

```
use std::collections::HashMap;

let mut mapa1 = HashMap::new();
mapa1.insert("a", 1);
mapa1.insert("b", 2);

let mut mapa2 = HashMap::new();
mapa2.insert("b", 2);
mapa2.insert("a", 1);

println!("{}", mapa1 == mapa2); // true (orden no importa)

let mut mapa3 = HashMap::new();
mapa3.insert("a", 1);
mapa3.insert("b", 3);

println!("{}", mapa1 == mapa3); // false
```

Rust

HashMap con structs personalizados

```
use std::collections::HashMap;

#[derive(Debug)]
struct Persona {
    nombre: String,
    edad: u32,
}
```

Rust

```
let mut personas = HashMap::new();

personas.insert(
    "001",
    Persona {
        nombre: String::from("Ana"),
        edad: 25,
    },
);

personas.insert(
    "002",
    Persona {
        nombre: String::from("Luis"),
        edad: 30,
    },
);

if let Some(persona) = personas.get("001") {
    println!("{:?}", persona);
}
```

Conversión a Vec

De HashMap a Vec de tuplas:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);
mapa.insert("c", 3);

let vec: Vec<(&str, i32)> = mapa.into_iter().collect();

println!("{:?}", vec);
// Orden no garantizado: [("a", 1), ("b", 2), ("c", 3)]
```

Rust

Solo claves a Vec:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);

let claves: Vec<&str> = mapa.keys().copied().collect();
```

Rust


```
println!("{:?}", claves);
```

Solo valores a Vec:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("a", 1);
mapa.insert("b", 2);

let valores: Vec<i32> = mapa.values().copied().collect();

println!("{:?}", valores);
```

Rust

Ordenar HashMap

HashMap NO mantiene orden. Para ordenar, convierte a Vec:

```
use std::collections::HashMap;

let mut mapa = HashMap::new();
mapa.insert("c", 3);
mapa.insert("a", 1);
mapa.insert("b", 2);

let mut items: Vec<_> = mapa.iter().collect();
items.sort_by_key(|&(k, _)| k); // ordenar por clave

for (clave, valor) in items {
    println!("{:?}", clave, valor);
}

// a: 1
// b: 2
// c: 3
```

Rust

Si necesitas mantener orden de inserción, usa **IndexMap** (requiere crate externo).

HashMap anidados

```
use std::collections::HashMap;

let mut usuarios = HashMap::new();

let mut info_ana = HashMap::new();
info_ana.insert("edad", "25");
info_ana.insert("ciudad", "Lima");
```

Rust

```
let mut info_luis = HashMap::new();
info_luis.insert("edad", "30");
info_luis.insert("ciudad", "Bogotá");

usuarios.insert("Ana", info_ana);
usuarios.insert("Luis", info_luis);

if let Some(info) = usuarios.get("Ana") {
    if let Some(edad) = info.get("edad") {
        println!("Ana tiene {} años", edad);
    }
}
```

Clonar HashMap

```
use std::collections::HashMap;

let mut mapa1 = HashMap::new();
mapa1.insert("a", 1);
mapa1.insert("b", 2);

let mapa2 = mapa1.clone();

println!("{:?}", mapa1); // {"a": 1, "b": 2}
println!("{:?}", mapa2); // {"a": 1, "b": 2}
```

Rust

△ `clone()` hace una copia profunda, puede ser costoso para HashMaps grandes.

Cuándo usar HashMap vs otros tipos

Usa `HashMap<K, V>` cuando:

- Necesitas búsquedas rápidas por clave
- Las claves son únicas
- No importa el orden
- Necesitas insertar/eliminar dinámicamente

Usa `BTreeMap<K, V>` cuando:

- Necesitas claves ordenadas
- Necesitas encontrar rangos de claves
- Iteras en orden frecuentemente

Usa `Vec<K, V>` cuando:

- Tienes pocas entradas (< 10-20)
- Necesitas mantener orden de inserción
- Iteras más que buscas

Usa `Array/Tupla` cuando:

- El número de elementos es fijo y pequeño
- Conoces todas las “claves” en compilación

Resumen rápido

```
use std::collections::HashMap;

// Crear
let mut mapa = HashMap::new();
let mapa = HashMap::from([("a", 1), ("b", 2)]);

// Insertar
mapa.insert("clave", valor);
mapa.entry("clave").or_insert(valor);

// Acceder
mapa.get("clave");           // Option<&V>
mapa["clave"];               // V (panic si no existe)
mapa.contains_key("clave");  // bool

// Actualizar
mapa.get_mut("clave");
mapa.entry("clave").and_modify(|v| *v += 1);

// Eliminar
mapa.remove("clave");
mapa.clear();
mapa.retain(|k, v| condición);

// Información
mapa.len();
mapa.is_empty();
mapa.capacity();

// Recorrer
for (k, v) in &mapa { }
for k in mapa.keys() { }
for v in mapa.values() { }

// Combinar
mapa1.extend(mapa2);
```

Rust

Control flow

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  
}
```

Rust

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");  
    }  
}
```

Rust

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
```

Rust

If expressions anidados También puedes encadenar múltiples condiciones:

```
let numero = 15;  
  
let resultado:&str = if numero < 0 {  
    "Negativo"
```

Rust

```
} 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
```

Rust

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)
```

Rust

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);
```

Rust

if con shadowing

```
let edad = 20;  
  
let edad = if edad >= 18 {
```

Rust

```
    "adulto"
  } else {
    "menor"
  };

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

if dentro de expresiones más grandes

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

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

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

Rust

Operadores lógicos

Rust soporta los operadores lógicos más comunes:

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

```
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");
}
```

Rust

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:

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

Rust

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

```
let mut contador = 0;  
  
loop {  
    contador += 1;  
    println!("Contador: {}", contador);  
  
    if contador == 5 {  
        break; // salimos del loop  
    }  
}
```

Rust

Explicación del código:

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
```

Output

loop con continue

‘continue’ permite saltar inmediatamente a la siguiente iteración del bucle.

```
let mut n = 0;  
  
loop {  
    n += 1;
```

Rust

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

println!("Número impar: {}", n);

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

- ‘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
```

Output

loop como expresión

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

Puedes usar ‘break’ para devolver un valor.

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

    let resultado = loop {
        contador += 1;

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

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

Rust

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:


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

Rust

- 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:

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

Rust

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

Ejemplo con label:

```
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.");  
}
```

Rust

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.
```

Output

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.");  
}
```

Rust

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
```

Output

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  
}
```

Rust

Ejemplo basico:

```
fn main() {  
    let mut contador = 0;  
  
    while contador < 5 {  
        println!("contador = {}", contador);  
        contador += 1; // importante: actualizar la condición en algún punto  
    }  
}
```

Rust

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
```

Output

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 {
```

Rust

```
        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 */ }

}
```

`nginx`

- `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"),
```

`Rust`

```
    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"),  
}
```

Rust

Patrón con múltiples alternativas

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

Rust

Patrones con variables

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

Rust

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"),  
}
```

Rust

Resultado:

Adulto

Output

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),  
    }  
}
```

Rust

Resultado:

```
Está en el eje Y, valor: 5
```

Output

Condiciones lógicas

```
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  
    }  
}
```

Rust

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:

```
let numero = 2;  
  
let texto = match numero {  
    1 => "uno",  
  
    2 => "dos",  
  
    3 => "tres",  
  
    _ => "desconocido",  
};
```

Rust

```
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)`:

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

Rust

Salida:

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

Output

Notas importantes sobre rangos:

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

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

Rust

Salida:

```
1  
2  
3
```

Output

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

For sobre arrays

Puedes iterar directamente sobre arrays:

```
let numeros = [10, 20, 30, 40, 50];  
  
for numero in numeros {  
    println!("Número: {}", numero);  
}
```

Rust

```
}
```

Salida:

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

Output

Iterar con referencia:

Si no quieres mover los valores, usa `&`:

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

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

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

Rust

For sobre vectores

Los vectores se iteran igual que los arrays:

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

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

Rust

Salida:

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

Output

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]
```

Rust

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);
}
```

Rust

Salida:

```
H
o
l
a
```

Output

Iterar sobre bytes:

```
let texto = "Rust";

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

Rust

Salida:

```
82
117
115
116
```

Output

△ 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);
}
```

Rust

Salida:

```
Hola
desde
Rust
```

Output

Iterar sobre líneas:

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

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

Rust

Salida:

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

Output

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);
}
```

Rust

Salida:

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

Output

Con vectores:

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

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

Rust

Salida:

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

Output

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 {
```

Rust

```
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
```

Output

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);  
}
```

Rust

Salida:

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

Output

For anidados

Puedes anidar bucles `for` :

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

Rust

Salida:

```
Posición (1, 1)
```

Output

```
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
}
```

Rust

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
```

Output

Iterar sobre arrays multidimensionales

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

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

Rust

Salida:

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

Output

Iterar en reversa

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

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

Rust

Salida:

```
5  
4  
3  
2  
1
```

Output

Con arrays:

```
let numeros = [10, 20, 30, 40, 50];  
  
for numero in numeros.iter().rev() {  
    println!("{}", numero);  
}
```

Rust

Salida:

```
50  
40  
30  
20  
10
```

Output

Iterar con pasos (step)

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

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

Rust

Salida:

```
0  
2  
4  
6  
8
```

Output

Casos comunes de for

Suma de elementos:

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

Rust

```
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;  
}
```

Rust

✓ Forma idiomática de Rust:

```
let arr = [10, 20, 30];  
  
for elemento in arr {  
    println!("{}", elemento);  
}
```

Rust

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
```

Rust

Prestar (con referencia):

```
let vec = vec![1, 2, 3];  
  
for item in &vec { // vec se presta
```

Rust

```
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]
```

Rust

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
```

Rust

Iteradores

El bucle `for` es uno de los más utilizados en Rust para iterar de manera segura y eficiente sobre rangos de valores o colecciones de datos. A diferencia de lenguajes como C, Rust no usa `for` con índices manuales esto para evitar errores comunes como `off-by-one`, sino que se basa en el `trait Iterator`. Esto hace que el código sea más idiomático, genérico y libre de errores de memoria.

Sintaxis básica:

```
for item in iterable {  
    // Código que usa `item` en cada iteración  
}
```

Rust

- **iterable**: Cualquier cosa que se pueda convertir en un iterador (rangos, vectores, arrays, etc.).
- **item**: Variable que recibe cada elemento por valor, referencia o desestructurada.
- Internamente, el `for` llama a `into_iter()` para crear un iterador y usa `next()` en un bucle `while let Some(...)` hasta que devuelve `None`.

El `for` toma `ownership` de la colección por defecto, pero puedes iterar por referencia “**&colección**” o mutable “**&mut colección**” para preservarla. Veremos esto en detalle más adelante.

Rangos

Los rangos son iteradores lazy “no crean arrays en memoria” y son perfectos para bucles numéricos.

Se definen con “`..`” exclusivo o “`..=`” inclusivo.

- **0..5**: Itera de 0 a 4.
- **0..=5**: Itera de 0 a 5.

Ejemplo básico:

```
fn main() {  
    for i in 0..5 {  
        println!("{}", i); // Salida: 0 1 2 3 4  
    }  
  
    for i in 0..=5 {  
        println!("{}", i); // Salida: 0 1 2 3 4 5  
    }  
}
```

Rust

- Reverso: Usa “**.rev()**” para iterar descendente.

```
fn main() {  
    for i in (0..=5).rev() {  
        println!("{}", i); // Salida: 5 4 3 2 1 0  
    }  
}
```

Rust

- Paso personalizado: “**.step_by(n)**” salta elementos.

```
fn main() {  
  
    for i in (0..10).step_by(2) {  
  
        println!("{}", i); // Salida: 0 2 4 6 8  
    }  
}
```

Rust

Los rangos implementan **IntoIterator** automáticamente. Son eficientes ($O(1)$ por elemento) y seguros (panic en debug si desbordan en modo checked).

Iterando sobre Colecciones: Arrays y Vectores

Las colecciones como arrays “**[T; N]**”, tamaño fijo y vectores “**Vec<T>**” dinámico, son iterables vía métodos como **into_iter()**, **iter()** y **iter_mut()**.

- **into_iter()**: Toma ownership y consume la colección (mueve elementos). Útil cuando quieres destruirla intencionalmente.
- **iter()**: Itera por referencias inmutables (&T). Preserva la colección.
- **iter_mut()**: Itera por referencias mutables (&mut T). Permite modificaciones sin consumir.

Iterando Arrays en Rust

En Rust, los bucles for permiten recorrer los elementos de un array de distintas maneras, dependiendo de cómo se accede a cada elemento:

- Por valor: se obtiene una copia del elemento.
- Por referencia: se accede sin mover ni copiar.
- Por referencia mutable: se accede para modificar los elementos.

Iteración por valor

Esta forma recorre los valores del array directamente.

```
fn main() {  
    let arr1 = [100, 200, 300];  
  
    println!("Iterando con for in (por valor):");  
    for x in arr1 {  
        println!("{}", x);  
    }  
  
    // Los arrays de tipos Copy (como i32) siguen disponibles  
    println!("Array original: {:?}", arr1);  
}
```

Rust

En arrays de tipos que implementan Copy, el array no se mueve. Pero en colecciones no Copy como “**Vec<String>**”, se perdería la propiedad.

Iteración con .into_iter()

Convierte el array en un iterador que consume sus elementos.

```
fn main() {
    let arr2 = [400, 500, 600];

    println!("Iterando con into_iter():");
    for x in arr2.into_iter() {
        println!("{}", x);
    }

    // En arrays de Copy (como i32) sigue disponible,
    // pero en vectores o structs NO.
    println!("Array original aún usable: {:?}", arr2);
}
```

Rust

“`into_iter()`” transfiere la propiedad de cada elemento al bucle.

Úsalo cuando ya no necesites el array después.

Iteración con `.iter()` por referencia

Se usa para prestar los valores sin moverlos.

```
fn main() {
    let arr3 = [700, 800, 900];

    println!("Iterando con iter():");
    for &x in arr3.iter() { // Desreferenciamos (&i32 → i32)
        println!("{}", x);
    }

    println!("Array original: {:?}", arr3);
}
```

Rust

“`iter()`” devuelve referencias `&T`, por lo que puedes leer los datos sin consumirlos.

Iteración con “`.iter_mut()`” por referencia mutable

Permite modificar los elementos del array durante la iteración.

```
fn main() {
    let mut arr4 = [1, 2, 3];

    println!("Iterando con iter_mut():");
    for x in arr4.iter_mut() {
        *x *= 10; // Desreferenciamos para modificar el valor
    }

    println!("Array modificado: {:?}", arr4);
}
```

Rust

- `iter_mut()` devuelve `&mut T`.
- Debes declarar el array como mut para usarlo.
- Es la única forma segura de modificar los valores dentro del bucle for.

Iterando Vectores en Rust

En Rust, los vectores `Vec<T>` se recorren con bucles `for` de distintas maneras, dependiendo de cómo se accede a los elementos:

- Por valor `for x in vec`, consume el vector.
- Por referencia `.iter()`, solo lectura.
- Por referencia mutable `.iter_mut()`, permite modificar.
- Por propiedad `.into_iter()`, transfiere la propiedad de los elementos.

Iteración por valor `for x in vec`

Cuando usamos `for x in vec`, Rust mueve el vector.

Después del bucle, ya no puedes usarlo.

```
fn main() {  
    let vec1 = vec![10, 20, 30];  
  
    println!("Iterando con for in <valor>:");  
    for x in vec1 {  
        println!("{}", x);  
    }  
  
    // Error si intentas usar vec1 después:  
    // println!("{:?}", vec1); // Moved value  
}
```

Rust

En vectores, a diferencia de los arrays, los valores no se copian automáticamente, se mueven.

Iteración con `.into_iter()`

`into_iter()` convierte el vector en un iterador que toma la propiedad de cada elemento.

```
fn main() {  
    let vec2 = vec!["Rust", "es", "genial"];  
  
    println!("→ Iterando con into_iter():");  
    for palabra in vec2.into_iter() {  
        println!("{}", palabra);  
    }  
  
    // El vector ya no está disponible:
```

Rust

```
// println!("{:?}", vec2); // Error
}
```

- Similar a `for x in vec`, pero más explícito.
- Útil cuando deseas consumir el vector completamente, por ejemplo al mover datos.

Iteración con `.iter()` por referencia

`iter()` presta referencias a los elementos del vector, sin moverlo.

```
fn main() {
    let vec3 = vec![100, 200, 300];

    println!("Iterando con iter():");
    for valor in vec3.iter() {
        println!("{}", valor);
    }

    println!("Vector original sigue disponible: {:?}", vec3);
}
```

Rust

- El tipo de valor es `&i32` “referencia”.
- Si quieres el valor directo, puedes desreferenciar: `for &valor in vec3.iter() { ... }`.

Iteración con `.iter_mut()` por referencia mutable

`iter_mut()` permite modificar los elementos directamente en el bucle.

```
fn main() {
    let mut vec4 = vec![1, 2, 3];

    println!("Iterando con iter_mut():");
    for valor in vec4.iter_mut() {
        *valor *= 10; // modificamos directamente
    }

    println!("Vector modificado: {:?}", vec4);
}
```

Rust

- El tipo de valor es `&mut i32`.
- Debes declarar el vector como `mut`.
- Los cambios se aplican directamente al vector original.

Iterar con índices cuando lo necesitas

A veces, también puedes iterar manualmente con índices si necesitas acceder a la posición.

```
fn main() {  
    let vec5 = vec!["A", "B", "C"];  
  
    println!("Iterando con índices:");  
    for i in 0..vec5.len() {  
        println!("Índice: {}, Valor: {}", i, vec5[i]);  
    }  
}
```

Rust

El acceso con `vec[i]` puede causar pánico si el índice está fuera del rango. Para evitarlo, usa `vec.get(i)` que devuelve `Option<T>`.

For con Tuplas

Una tupla en sí no es iterable.

No puedes hacer directamente:

```
for x in (1, 2, 3) { } // Error
```

Rust

Esto se debe a que una tupla es una estructura de longitud fija y heterogénea, no una secuencia.

Sin embargo, puedes iterar colecciones que contienen tuplas, como vectores o arrays, y desestructurarlas dentro del for.

Ejemplo:

```
let pares = vec![(1, "uno"), (2, "dos"), (3, "tres")];  
  
for (n, palabra) in pares {  
    println!("{}", n, palabra); // consume pares  
}
```

Rust

- `for (n, palabra)` in pares desestructura cada tupla del vector.
- El vector pares es consumido “moved” porque no usamos referencias.

Si queremos conservarlo:

```
let mapa = vec![(1, 2), (3, 4)];  
  
for (a, b) in &mapa { // iterando por referencia  
    println!("{}", a, b, a + b);  
}
```

Rust

Aquí, a y b son referencias `&i32`, y el vector mapa sigue disponible después del bucle.

For con Strings (&str y String)

Las cadenas de texto pueden recorrerse de distintas formas, según qué desees iterar: caracteres, bytes o graphemes (combinaciones visuales de caracteres).

Ejemplo:

```
let s = "hola 😊";

for c in s.chars() {
    println!("{}", c);
}

for b in s.bytes() {
    println!("{}", b);
}
```

Rust

Detalles importantes:

- `for c in s.chars()` , itera sobre caracteres Unicode (char).

Ejemplo de salida: `h` , `o` , `l` , `a` , , `😊` .

- `for b in s.bytes()` , itera sobre bytes **u8**, mostrando la codificación UTF-8.

Ejemplo: `104` , `111` , `108` , `97` , `32` , `240` , `159` , `152` , `138` .

break y continue

- `break` : termina el bucle actual.
- `continue` : salta a la siguiente iteración.

```
for i in 0..6 {
    if i == 3 {
        continue; // se salta el 3
    }
    if i == 5 {
        break; // rompe el bucle
    }
    println!("{}", i);
}
```

Rust

Etiquetas en For

Rust permite etiquetar bucles para romper o continuar desde un nivel específico.

```
'outer: for i in 1..=3 {
    for j in 1..=3 {
        if i == 2 && j == 2 {
            break 'outer; // rompe el bucle externo
        }
        println!("i={}, j={}", i, j);
    }
}
```

Rust

Salida:

```
i=1, j=1
i=1, j=2
i=1, j=3
i=2, j=1
```

Output

Ownership en for

Por defecto, `for` consume “mueve” la colección.

Para evitarlo, se itera por referencia `&` o `&mut`.

Ejemplo mutable:

```
fn main() {
    let mut vec = vec![1, 2, 3];
    for num in vec.iter_mut() { // Itera con mut borrow
        *num += 1;
    }
    println!("{:?}", vec); // [2, 3, 4]
}
```

Rust

Iteradores Avanzados

Los iteradores son una parte esencial de Rust: permiten recorrer datos de forma inmutable, segura y funcional, sin necesidad de bucles imperativos.

Cualquier tipo que implemente el trait **Iterator** puede usarse en un **for**, o manipularse mediante sus métodos.

Creación de iteradores

```
let v = vec![10, 20, 30];
let mut it = v.iter(); // &i32

println!("{:?}", it.next()); // Some(&10)
println!("{:?}", it.next()); // Some(&20)
println!("{:?}", it.next()); // Some(&30)
println!("{:?}", it.next()); // None
```

Rust

Cada llamada a `.next()` devuelve `Option<T>`, hasta que se agotan los elementos.

Métodos comunes:

- `map()` : transforma

```
let v = vec![1, 2, 3];
let doblados: Vec<_> = v.iter().map(|x| x * 2).collect();
println!("{:?}", doblados); // [2, 4, 6]
```

Rust

- `filter()` : filtra según condición

```
let v = vec![1, 2, 3, 4, 5];
let pares: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

Rust


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

- **fold()** : acumula (reduce)

```
let suma = (1..=5).fold(0, |acc, x| acc + x);  
println!("{}", suma); // 15
```

Rust

- **take()** **skip()**

```
let v = vec![10, 20, 30, 40, 50];  
  
for n in v.iter().skip(1).take(3) {  
    println!("{}", n);  
}  
  
// Imprime: 20, 30, 40
```

Rust

- **rev()** : invierte el orden

```
for n in (1..=3).rev() {  
    println!("{}", n);  
}  
  
// 3, 2, 1
```

Rust

Combinaciones poderosas

```
let datos = vec![1, 2, 3, 4, 5, 6];  
  
let resultado: Vec<_> = datos  
    .iter()  
    .filter(|x| *x % 2 == 0) // pares  
    .map(|x| x * x)          // al cuadrado  
    .rev()                   // orden inverso  
    .collect();  
  
println!("{:?}", resultado); // [36, 16, 4]
```

Rust

Iterar múltiples colecciones

```
fn main() {  
    let a = vec![1, 2, 3];  
  
    let b = vec!["uno", "dos", "tres"];  
  
    for (num, texto) in a.iter().zip(b.iter()) { // Tupla (i32, &str)  
        println!("{: }: {}", num, texto);  
    }  
  
    // Salida: 1: uno, 2: dos, 3: tres  
    // Se detiene en el más corto.  
}
```

Rust

Búsqueda de Iteradores

Estos métodos trabajan sobre iteradores y devuelven valores booleanos o resultados parciales en lugar de colecciones completas.

No consumen todos los elementos si no es necesario, paran cuando encuentran el resultado.

- **any()**

Devuelve true si al menos un elemento cumple la condición.

Es equivalente a decir:

“¿Existe alguno que cumpla esto?”

```
fn main() {  
    let numeros = vec![1, 2, 3, 4, 5];  
  
    let hay_par = numeros.iter().any(|&x| x % 2 == 0);  
    println!("¿Hay algún número par? {}", hay_par); // true  
  
    let mayor_a_diez = numeros.iter().any(|&x| x > 10);  
    println!("¿Alguno mayor a 10? {}", mayor_a_diez); // false  
}
```

Rust

any() deja de iterar en cuanto encuentra un true.

Ejemplo: no sigue buscando más pares cuando ya encontró uno.

- **all()**

Devuelve true si todos los elementos cumplen la condición.

Es equivalente a decir:

“¿Todos cumplen esto?”

```
fn main() {  
    let edades = vec![18, 22, 25, 30];  
  
    let todos_mayores = edades.iter().all(|&x| x >= 18);  
    println!("¿Todos son mayores de edad? {}", todos_mayores); // true  
  
    let todos_pares = edades.iter().all(|&x| x % 2 == 0);  
    println!("¿Todos son pares? {}", todos_pares); // false  
}
```

Rust

all() también detiene la iteración en cuanto encuentra un false.

- **find()**

Busca el primer elemento que cumpla una condición y devuelve Option.

Es equivalente a decir:

“Encuentra el primer elemento que cumpla esto”.

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

    if let Some(x) = numeros.iter().find(|&n| n > 25) {
        println!("El primero mayor a 25 es: {}", x); // 30
    } else {
        println!("No hay ninguno mayor a 25");
    }

    // Otro ejemplo: buscar una palabra
    let palabras = vec!["rust", "golang", "python"];

    match palabras.iter().find(|&p| p.starts_with('p')) {
        Some(p) => println!("Encontrado: {}", p), // python
        None => println!("No encontrado"),
    }
}
```

Rust

- `find()` devuelve una referencia `Option<T>` si usas `.iter()`.
- Si usas `.into_iter()`, moverá los elementos `Option<T>`.

Ejemplo combinado

Estos métodos se pueden combinar para expresiones potentes:

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];

    // ¿Existe algún impar mayor a 3?
    let resultado = nums.iter().any(|&x| x % 2 == 1 && x > 3);
    println!("¿Impar > 3?: {}", resultado); // true (5)

    // Verificar todos menores que 10
    println!("¿Todos < 10?: {}", nums.iter().all(|&x| x < 10)); // true

    // Buscar primer número divisible por 4
    if let Some(x) = nums.iter().find(|&x| x % 4 == 0) {
        println!("Divisible por 4: {}", x); // 4
    }
}
```

Rust

Concepto clave

Los iteradores en Rust son lazy (perezosos): no ejecutan nada hasta que se consumen, por ejemplo con:

- `collect()`
- `for`

- count()
- sum()
- any(), all(), find()

Ejemplo:

```
let v = vec![1, 2, 3];
let it = v.iter().map(|x| x * 10); // aún no se ejecuta
println!("{}", it.sum:<i32>()); // se ejecuta aquí
```

Rust

Detallar los iteradores con javascript!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Funciones en Rust

Las funciones son bloques de código reutilizables diseñados para realizar una tarea específica. Facilitan la modularidad, el mantenimiento y la reutilización del código al dividir un programa en partes más pequeñas y manejables. En Rust, las funciones se distinguen de los métodos que están asociados a structs o enums mediante impl, aquí nos enfocamos en funciones independientes.

Se declaran con la palabra clave `fn`, y Rust enfatiza la seguridad de tipos: los parámetros siempre requieren tipos explícitos, y los retornos son opcionales pero tipados.

Sintaxis básica:

```
fn nombre_funcion(param1: Tipo1, param2: Tipo2) -> TipoRetorno {
    // Cuerpo de la función
    // El último expresión sin ; se devuelve automáticamente
}
```

Rust

Ejemplo simple:

```
fn saludar() {
    println!("¡Hola, mundo!");
}
fn main() {
    saludar(); // Llama la función
}
```

Rust

Parámetros

Los parámetros son entradas a la función y deben especificar su tipo. Rust no infiere tipos en declaraciones de parámetros para evitar ambigüedades.

```
/// Suma dos números enteros y los imprime.
fn sumar(a: i32, b: i32) {
    println!("La suma es: {}", a + b);
}

fn main() {
    sumar(5, 3); // Salida: La suma es: 8
}
```

Rust

Valores de retorno

Las funciones pueden devolver valores. Si no se especifica, devuelven “`() (unit type)`”.

- Retorno implícito:

La última expresión sin “`;`” se devuelve automáticamente, estilo idiomático de Rust.

```
fn cuadrado(x: i32) -> i32 {  
    x * x // Sin ";" devuelve el resultado  
}  
  
fn main() {  
    let resultado = cuadrado(4);  
    println!("Cuadrado: {}", resultado); // 16  
}
```

Rust

Si agregas “`;`”:

```
fn cuadrado_con_semi(x: i32) -> i32 {  
    x * x; // Con ";" devuelve ()  
    0      // Esto se ignora  
}
```

Rust

return Explícito

Para salidas tempranas “early returns”, usa “`return expr;`”. Es opcional al final, pero útil en condicionales.

```
fn main() {  
    let a = 15;  
    let b = 9;  
  
    println!("El valor mayor de {a} y {b} es : {}", valor_mayor(a , b));  
}  
  
fn valor_mayor(a:i32, b:i32) -> i32{  
    if a > b {  
        return a  
    }else{  
        return b  
    }  
}
```

Rust

Otro ejemplo: Verificar paridad.

```
fn es_par(n: i32) -> bool {  
    if n % 2 == 0 {  
        true // Expresión
```

Rust

```
    } else {  
        false  
    }  
}  
  
/// Equivalente con return (más conciso).  
fn es_par_ret(n: i32) -> bool {  
    return n % 2 == 0; // Explícito  
}
```

Funciones anidadas

Rust permite definir funciones dentro de otras “**inner functions**”, pero solo visibles en el scope padre. Útiles para helpers locales.

```
fn principal() {  
    fn interna(x: i32) -> i32 { // Anidada  
        x + 1  
    }  
    println!("{}", interna(5)); // 6  
}  
  
fn main() {  
    principal();  
}
```

Rust

Closures: Funciones Anónimas

Los closures son funciones anónimas que capturan el entorno “**variables externas**”. Son como lambdas en Python/JS, pero con reglas estrictas de captura “**borrow/move**”. Ideales para iteradores, callbacks y FP.

Sintaxis:

```
let nombre_closure = |param1: Tipo1, param2: Tipo2| -> TipoRetorno {  
    // Cuerpo (expresión o bloque)  
};
```

Rust

Ejemplo básico:

```
let resta = |a: i32, b: i32| -> i32 {a - b}; // Explícito  
let suma = |a: i32, b: i32| a + b; // Infieren tipos si posible  
println!("{}", suma(2, 3)); // 5  
println!("{}", resta(5, 3)); // 2
```

Rust

Devolviendo Múltiples Valores: Tuplas

Las funciones pueden devolver múltiples valores empaquetados en una tupla:

```
fn calcular(a: i32, b: i32) -> (i32, i32) {
    (a + b, a * b) // Tupla como último valor
}

fn main() {
    let (suma, producto) = calcular(4, 5);
    println!("Suma: {}, Producto: {}", suma, producto); // 9, 20
}
```

Rust

Devolver un array

Los arrays en Rust tienen tamaño fijo que se define en el tipo.

```
fn crear_array() -> [i32; 3] {
    [10, 20, 30]
}

fn main() {
    let arr = crear_array();
    println!("Array: {:?}", arr); // [10, 20, 30]
}
```

Rust

Devolver un vector

Un vector `Vec<T>` es dinámico y mucho más usado en la práctica:

```
/// Crea un vector de strings.
fn crear_vector() -> Vec<String> {
    vec!["Rust".to_string(), "es".to_string(), "genial".to_string()]
}

fn main() {
    let v = crear_vector();
    println!("Vector: {:?}", v); // ["Rust", "es", "genial"]
}
```

Rust

Funciones con referencias

Pasa datos grandes por `&T` para evitar copias/moves.

```
/// Calcula longitud sin tomar ownership.
fn longitud(texto: &String) -> usize { // &String (o &str para más flex)
    texto.len()
}

fn main() {
    let s = String::from("Hola Rust");
}
```

Rust

```
let l = longitud(&s); // Borrow
println!("{}", tiene {} caracteres", s, l); // s intacto
}
```

Funciones por referencia y mutables

```
fn incrementar(x: &mut i32) {
    *x += 1; // se desreferencia con *
}

fn main() {
    let mut num = 10;
    incrementar(&mut num);
    println!("{}", num); // 11
}
```

Rust

Explicación rápida:

- “`&mut`”, referencia mutable (permite modificar la variable original).
- “`*x`”, desreferencia: accede al valor real dentro de la referencia.

Ejemplos prácticos de funciones con `&mut`

Ahora veamos casos más útiles y aplicables.

Ejemplo 1: Modificar un vector desde una función

Supongamos que queremos agregar elementos a un vector existente.

```
fn agregar_elemento(v: &mut Vec<i32>, nuevo: i32) {
    v.push(nuevo);
}

fn main() {
    let mut numeros = vec![1, 2, 3];
    agregar_elemento(&mut numeros, 4);
    agregar_elemento(&mut numeros, 5);

    println!("{:?}", numeros); // [1, 2, 3, 4, 5]
}
```

Rust

Aquí:

- Modifica directamente el vector original gracias a la referencia mutable.
- Sin `&mut`, el `push()` fallaría porque Rust no permitiría modificarlo.

Ejemplo 2: Convertir texto a mayúsculas

```
fn a_mayusculas(texto: &mut String) {
```

Rust


```
*texto = texto.to_uppercase();
}

fn main() {
    let mut saludo = String::from("hola rust");
    a_mayusculas(&mut saludo);
    println!("{}", saludo); // HOLA RUST
}
```

Explicación:

- `to_uppercase()` crea un nuevo `String`.
- Lo reasignamos sobre el mismo valor usando “`*texto =`”

Ejemplo 3: Restablecer valores en un arreglo mutable

```
fn resetear(arr: &mut [i32]) {
    for x in arr.iter_mut() {
        *x = 0;
    }
}

fn main() {
    let mut datos = [3, 7, 9];
    resetear(&mut datos);
    println!("{:?}", datos); // [0, 0, 0]
}
```

Rust

Puntos clave:

- `&mut [i32]` es un slice mutable del array.

Ejemplo 4: Intercambiar valores (swap)

```
fn intercambiar(a: &mut i32, b: &mut i32) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut x = 10;
    let mut y = 20;
    intercambiar(&mut x, &mut y);
    println!("x = {}, y = {}", x, y); // x = 20, y = 10
}
```

Rust

Útil cuando quieres manipular variables sin devolver múltiples valores.

Funciones genéricas

Una función genérica permite que el tipo de sus parámetros no esté fijo.

En lugar de hacer una función solo para `i32` o `f64`, puedes hacer una que funcione con cualquier tipo.

Ejemplo sin genéricos

```
fn mostrar_i32(x: i32) {  
    println!("El valor es: {}", x);  
}  
  
fn main() {  
    mostrar_i32(10);  
}
```

Rust

Esta función solo acepta `i32`.

Si intentas pasarle un `f64`, no compila.

Ejemplo genérico básico

```
fn mostrar<T>(x: T) {  
    println!("Tengo un valor!");  
}  
  
fn main() {  
    mostrar(10);           // i32  
    mostrar(3.14);         // f64  
    mostrar("hola");       // &str  
}
```

Rust

Retorno genérico

```
fn devolver_mismo<T>(x: T) -> T {  
    x  
}  
  
fn main() {  
    let n = devolver_mismo(5);  
  
    let t = devolver_mismo("texto");  
  
    println!("{}", n, t);  
}
```

Rust

Esta función devuelve exactamente el mismo tipo que recibe.

Ejemplo 1: Intercambiar valores

Una función genérica puede manipular cualquier tipo, no solo devolverlo.

Veamos cómo intercambiar dos valores de cualquier tipo (sin importar si son `i32`, `String`, etc.):

```
fn intercambiar<T>(a: &mut T, b: &mut T) {  
    let temp = *a;    // mueve el valor de a  
    *a = *b;          // reemplaza a por b  
    *b = temp;        // reemplaza b por temp  
}  
  
fn main() {  
    let mut x = 10;  
    let mut y = 20;  
  
    intercambiar(&mut x, &mut y);  
    println!("x = {}, y = {}", x, y); // x = 20, y = 10  
}
```

Rust

Aquí:

- `T` puede ser cualquier tipo (numérico, texto, struct...).
- `&mut T` indica que tomamos referencias mutables (podemos modificarlas).
- No necesitamos saber qué tipo es `T`, Rust se encarga en tiempo de compilación.

Ejemplo 4: Genéricos + funciones que retornan otro tipo genérico

```
fn convertir_en_vec<T>(x: T) -> Vec<T> {  
    vec![x]  
}  
  
fn main() {  
    let numeros = convertir_en_vec(42);  
    let palabras = convertir_en_vec("hola");  
  
    println!("{:?}", numeros); // [42]  
    println!("{:?}", palabras); // ["hola"]  
}
```

Rust

Esta función:

- Toma un valor de tipo `T`
- Devuelve un `Vec<T>` (vector del mismo tipo)

Struct

En Rust, un struct **‘estructura’** es una forma de agrupar múltiples valores relacionados bajo un tipo de dato unificado. Funciona como un contenedor organizado que nos permite definir nuestro propio tipo de dato personalizado.

Por ejemplo, imagina un Libro como una ficha bibliográfica: en lugar de manejar datos sueltos sobre un libro, reunimos toda la información relevante como el título, autor y año de publicación en un solo registro con campos específicamente etiquetados.

Esto no solo mantiene los datos lógicamente unidos, sino que también le da al compilador de Rust la capacidad de verificar que accedemos a campos válidos, como título o autor. Si intentas acceder a un campo no definido, como editorial, Rust te avisará de ese error en tiempo de compilación.

Además, al asignar tipos correctos a cada campo como por ejemplo, String para el título y u32 para el año, prevenimos errores comunes antes de que el programa se ejecute, en lugar de descubrirlos durante la ejecución. Esto hace que el código sea más robusto y fácil de mantener.

Sintaxis básica

Un struct se declara con la palabra clave `struct`, seguida del nombre y una lista de campos entre llaves:

```
struct Libro {  
    titulo: String,  
    autor: String,  
    anio_publicacion: u32,  
}
```

Rust

- `Libro` es el nombre del `struct`.
- `titulo`, `autor` y `anio_publicacion` son los campos.
- Cada campo tiene un nombre y un tipo.
- Los nombres deben usar `PascalCase`, según la convención oficial de Rust.

Instanciación de un Struct

Para crear una instancia del `struct`, se usa las llaves `{ }` indicando los valores de cada campo.

```
fn main() {  
    let libro = Libro {  
        titulo: String::from("Domina Rust"),  
        autor: String::from("Alex Villanueva"),  
        anio_publicacion: 2025,  
    };  
}
```

Rust

Reglas importantes:

- Debes inicializar todos los campos.
- El orden no importa.
- Puedes crear structs como `let` o `let mut`.

Acceso a los campos

Los campos se acceden con el operador `.`:

```
// --snip--
```

Rust

```
println!("Título: {}", libro.titulo); // "Domina Rust"
println!("Autor: {}", libro.autor);    // "Alex Villanueva"
println!("Año: {}", libro.anio_publicacion); // 2025
```

Structs Mutables

Para modificar los campos de un struct, la instancia debe ser mutable `mut`:

```
let mut libro = Libro {
    // --snip--
};

libro.anio_publicacion = 2026; // Modificamos un campo
println!("Año: {}", libro.anio_publicacion); // Ahora 2026
```

Rust

Si la variable no es mutable, sus campos tampoco lo serán.

Struct Update Syntax

Puedes crear una nueva instancia copiando algunos campos de otro struct con la sintaxis “`..`”:

```
// --snip--

let libro2 = Libro {
    titulo: String::from("Aprende RusT"),
    ..libro
};

println!("{}", libro2.titulo); // "Aprende RusT"
println!("{}", libro2.autor);  // "Alex Villanueva"
println!("{}", libro2.anio_publicacion); // 2026
```

Rust

Esto copia:

- libro.autor
- libro.anio_publicacion

Si un campo contiene datos que mueven ownership como `String`, la variable original deja de ser válida para ese campo.

Tuple Structs

Son structs sin nombre para los campos. Útiles cuando el significado viene implícito:

```
struct Punto(i32, i32);

let p = Punto(10, 20);
```

Rust

Unit-like Structs

No tienen campos. Se usan para:

- Implementar traits en tipos sin datos
- Marcar eventos o comportamientos

- Crear tipos fantasma (zero-sized types)

```
struct SinDatos;  
  
let x = SinDatos;
```

Rust

Desestructuración de Structs

Podemos extraer los campos de un struct en variables individuales:

```
// --snip--  
  
let Libro { titulo, autor, anio_publicacion } = libro2;  
println!("{}", - {} ({}), titulo, autor, anio_publicacion);
```

Rust

Derivaciones automáticas

Rust puede generar implementaciones comunes automáticamente con `#[derive(...)]`.

```
#[derive(Debug, Clone, PartialEq)]  
struct Libro {  
    // --snip--  
}  
  
fn main() {  
    // --snip--  
    let Libro { titulo, autor, anio_publicacion } = libro2.clone();  
    // --snip--  
    println!("{}", libro); // Usa Debug  
    println!("{}", libro); // Usa Debug  
  
    let libro3 = Libro{  
        titulo: String::from("EFFECTIVE RUST"),  
        autor: String::from("David Drysdale"),  
        anio_publicacion: 2000,  
    };  
  
    println!("{}", libro2 == libro3); // False  
}
```

Rust

- `Debug` : permite imprimir structs con `{:?}`.
- `Clone` : permite clonar valores.
- `PartialEq` : permite comparar con `==`.

Métodos en Structs

En Rust, cuando queremos añadir comportamiento a nuestros structs, utilizamos el bloque `impl` para implementar métodos.

Esto transforma nuestros structs de ser simples contenedores de datos estaticos a tipos de datos inteligentes que saben cómo operar con su propia información, manteniendo todo el código

relacionado organizado en un mismo lugar y aprovechando el sistema de tipos de Rust para garantizar que estas operaciones se realicen de manera segura.

Sintaxis:

```
impl Libro {  
    // Aquí van los métodos y funciones asociadas  
}
```

Rust

Ejemplo básico:

```
// --snip--  
impl Libro {  
    fn new(titulo: String, autor: String, anio_publicacion: u32) -> Libro {  
        Libro {  
            titulo,  
            autor,  
            anio_publicacion  
        }  
    }  
}  
  
fn main() {  
    // --snip--  
    let libro4 = Libro::new(  
        String::from("Domina Rust"),  
        String::from("Alex Villanueva"),  
        2025  
    );  
    // Usa :: para métodos asociados  
}
```

Rust

Self

Self representa el tipo del `struct` dentro del bloque `impl`.

Es una forma abreviada de escribir el nombre completo del tipo. `Self = Libro`.

```
fn new(..) -> Self {  
    Self {  
        ...  
    }  
}
```

Rust

```
fn new(..) -> Libro {  
    Libro {  
        ...  
    }  
}
```

Rust

Las Variantes de self

Todos los métodos de instancia usan `self` como primer parámetro. Las variantes controlan ownership y mutabilidad. En Rust, los métodos dentro de `impl` pueden recibir al propio objeto de diferentes formas.

1. Borrow Inmutable `self`

- Referencia inmutable al objeto actual.
- No puede modificar los campos, solo leerlos.
- Se traduce a: `self: &Self`.

Ejemplo:

```
impl Libro {  
    fn es_reciente(&self) -> bool { // &self: solo lee  
        self.ano_publicacion >= 2020  
    }  
}
```

Rust

Forma abreviada de escribir:

```
fn es_reciente(&self) -> fn es_reciente(self: &Libro)
```

Output

```
fn main() {  
    // --snip--  
  
    let libro4 = Libro::nuevo(  
        String::from("Nuevo Libro"),  
        String::from("Autor X"),  
        2025  
    );  
  
    println!("¿Es reciente? {}", libro4.es_reciente()); // true  
    // libro sigue intacto, puedes usarlo después  
    println!("Año: {}", libro4.ano_publicacion); // 2026  
}
```

Rust

2. Borrow Mutable `&mut self`

- Modifica y lee los campos.
- Solo una mutación a la vez.

Ejemplo:

```
impl Libro {  
    fn actualizar_año(&mut self, nuevo_año: u32) { // &mut self: modifica  
        self.ano_publicacion = nuevo_año;  
    }  
}
```

Rust

Forma abreviada de escribir:

```
fn actualizar_año(&mut self, ...) -> fn actualizar_año(self: &mut Libro, ...)
```

Necesitas

- `let mut libro`
- `&mut self`.

```
fn main() {
    // --snip--

    libro4.actualizar_año(2027); // Cambia vía método
    println!("Año actualizado: {}", libro.anio_publicacion); // 2027
    // libro mut sigue vivo y modificado
}
```

Rust

3. Tomar Ownership `self`

Mueve la instancia al método **ownership transferido**.

Después, no puedes usar el struct original.

Ejemplo:

```
impl Libro {

    fn a_string(self) -> String { // self: consume
        format!("{}", por {} ({}),
            self.titulo,
            self.autor,
            self.anio_publicacion
        )
    }
}

fn main() {
    // --snip--

    let descripcion = libro4.a_string(); // Consume libro4
    println!("{}", descripcion); // Domina Rust por Alex Villanueva (2027)
    //println!("{:?}", libro4); // Error: moved!
}
```

Rust

4. `&Self` & `&mut Self`

Se usan normalmente en las firmas de métodos estáticos o asociados, o cuando defines tipos o traits genéricos, donde `Self` hace que tu código sea más genérico y expresivo.

```
impl Libro{

    fn comparar(&self, otro: &Self) -> bool { // &Self = &Libro
        self.titulo == otro.titulo
    }
}
```

Rust

```
fn incrementar(&mut self, otro: &mut Self) {
    otro.anio_publicacion += self.anio_publicacion;
}
}
```

Uso:

```
fn main(){
    // --snip--

    libro_y.incrementar(&mut libro_x);
    println!("{}", libro_y.anio_publicacion);

    println!("{}", libro_y.comparar(&libro_x));
}
```

Rust

Method Chaining

En Rust, es posible llamar varios métodos de forma consecutiva sobre el mismo valor usando un estilo fluido conocido como encadenamiento de métodos “method chaining”.

Este patrón es muy común en APIs modernas, ya que hace el código más expresivo y fácil de leer.

```
persona
    .incrementar(1)
    .incrementar(2)
    .resetear();
```

Rust

Este estilo solo es posible cuando los métodos están implementados dentro de un bloque `impl`.

¿Qué permite el encadenamiento de métodos?

Para encadenar métodos, un método debe retornar `Self`, `&Self` o `&mut Self`.

Es decir, debe devolver la propia instancia para poder llamar al siguiente método sobre ella.

Rust no tiene un “azúcar sintáctico especial” para esto; es solo una consecuencia natural de devolver el mismo objeto.

Ejemplo básico:

```
struct Persona {
    edad: u8,
}

impl Persona {
    fn new(edad: u8) -> Self {
        Self { edad }
    }

    // Permite encadenar gracias a Self
    fn sumar(mut self, x: u8) -> Self {
```

Rust

```
        self.edad += x;
        self
    }

    // También retorna Self
    fn duplicar(mut self) -> Self {
        self.edad *= 2;
        self
    }

    fn mostrar(&self) {
        println!("Edad: {}", self.edad);
    }
}

fn main() {
    let persona = Persona::new(20)
        .sumar(5)      // 25
        .duplicar();  // 50

    persona.mostrar();
}
```

Salida:

Edad: 50

Rust

Enums y Patrones

Rust ofrece una poderosa combinación entre enumeraciones “enums” y el sistema de patrones “pattern matching”.

Juntos permiten representar datos complejos de forma segura y expresiva.

Enum

Un enum te permite definir un tipo que puede ser uno entre varios valores posibles.

Ejemplo basico:

```
enum Color {
    Rojo,
    Verde,
    Azul,
}
```

Rust

Los enum son muy útiles para representar estados, opciones y variaciones de datos.

Enums con Datos

A diferencia de otros lenguajes, Rust permite que cada variante del enum tenga datos asociados.

- Con valores simples

```
enum Ip {  
    V4(String),  
    V6(String),  
}
```

Rust

- Con estructuras diferentes

```
enum Mensaje {  
    Texto(String),  
    Coordenada { x: i32, y: i32 },  
    Salir,  
}
```

Rust

match: La herramienta principal para enums

match permite comparar un valor contra patrones.

Ejemplo básico:

```
fn imprimir_color(color: Color) {  
    match color {  
        Color::Rojo => println!("Rojo!"),  
        Color::Verde => println!("Verde!"),  
        Color::Azul => println!("Azul!"),  
    }  
}
```

Rust

Patrones con Enums que tienen datos

- Extraer valores

```
enum Mensaje {  
    Texto(String),  
    Numero(i32),  
}  
  
fn procesar(m: Mensaje) {  
    match m {  
        Mensaje::Texto(t) => println!("Texto: {}", t),  
        Mensaje::Numero(n) => println!("Número: {}", n),  
    }  
}
```

Rust

- Con estructura

```
enum Comando {  
    Mover { x: i32, y: i32 },  
}  
  
fn ejecutar(c: Comando) {  
    match c {
```

Rust

```
Comando::Mover { x, y } => {  
    println!("Moviendo a la posición ({}, {})", x, y);  
}  
}  
}
```

Patrones con rangos

```
enum Evento {  
    Nivel(i32),  
}  
  
match Evento::Nivel(5) {  
    Evento::Nivel(n @ 1..=10) => println!("Nivel bajo: {}", n),  
    Evento::Nivel(n) => println!("Nivel alto: {}", n),  
}
```

Rust

Enums con Métodos

Los enums también pueden tener métodos.

```
enum Estado {  
    Online,  
    Offline,  
}  
  
impl Estado {  
    fn esta_activo(&self) -> bool {  
        matches!(self, Estado::Online)  
    }  
}
```

Rust

Uso:

```
let e = Estado::Online;  
  
println!("{}", e.esta_activo()); // true
```

Rust

Imprimir Enum

```
#[derive(Debug)]  
enum Direccion {  
    Norte,  
    Sur,  
    Este,  
    Oeste,  
}  
  
fn main() {  
    let dir = Direccion::Norte;
```

Rust

```
println!("{:?}", dir);
}
```

Generic Types

Los genéricos permiten escribir código que funciona con múltiples tipos sin duplicar lógica. Son fundamentales en Rust porque habilitan:

- Reutilización de código
- Flexibilidad sin perder seguridad
- Rendimiento máximo gracias a la monomorfización

¿Qué son los genéricos?

Un generic type es un “comodín de tipos” que le permite al compilador crear múltiples versiones de una misma estructura, enum o función, una por cada tipo concreto que uses, sin que tengas que escribir el código una y otra vez.

- En lugar de decir:

“Esta estructura solo sirve para i32”

- Dices:

“Esta estructura sirve para cualquier tipo y le pondré de nombre T y tú como compilador, generas la versión correcta cuando se use con i32, String, User, etc.”

Sintaxis:

```
fn identidad<T>(valor: T) -> T {
    valor
}
```

Rust

`<T>` es un parámetro de tipo genérico. Esto significa que la función funciona para:

- `i32`, `String`, `bool` etc
- incluso para tipos definidos por el usuario.

Ejemplo:

```
let n = identidad(10);
let s = identidad("hola");
let b = identidad(true);
```

Rust

Rust genera una versión optimizada para cada tipo utilizado **monomorfización**.

Genéricos en estructuras

Puedes declarar un struct que acepte tipos genéricos:

```
struct Punto<T> {
    x: T,
    y: T,
}
```

Rust

Uso:

```
let p1 = Punto { x: 10, y: 20 };
let p2 = Punto { x: 3.5, y: 6.1 };
let p3 = Punto { x: "hola", y: "mundo" };
```

Rust

Incluso diferentes tipos en un mismo struct:

```
struct Par<T, U> {
    a: T,
    b: U,
}

let par = Par { a: "edad", b: 24 };
```

Rust

Métodos genéricos `impl<T>`

También puedes definir métodos dentro de un impl genérico:

```
struct Contenedor<T> {
    valor: T,
}

impl<T> Contenedor<T> {
    fn obtener(&self) -> &T {
        &self.valor
    }
}
```

Rust

Incluso métodos adicionales para un tipo concreto:

```
impl Contenedor<i32> {
    fn es_par(&self) -> bool {
        self.valor % 2 == 0
    }
}
```

Rust

Funciones con múltiples genéricos

Una función puede tener dos, tres o más tipos genéricos, cada uno con sus propias restricciones.

```
fn mezclar<T, U, V>(a: T, b: U, c: V) -> (T, U, V) {
    (a, b, c)
}
```

Rust

Traits

Un trait es un conjunto de métodos que un tipo puede implementar. Es similar a interfaces en otros lenguajes, pero más poderoso.

Sirve para:

- Definir comportamientos comunes para distintos tipos.
- Permitir polimorfismo.

- Exigir que un tipo tenga cierto método para poder usarse en una función genérica.
- Poder escribir código genérico sin importar el tipo mientras cumpla el trait.

Traits Comunes de la Librería Estándar

Antes de crear tus propios traits, es importante conocer algunos de los traits más utilizados que vienen incluidos en la Standard Library.

Muchos tipos los implementan automáticamente y aparecen con frecuencia en código real.

1. Trait Debug

Permite imprimir un valor para depuración con `{:?}` o `{:#?}`.

```
#[derive(Debug)]
struct Persona {
    nombre: String,
    edad: u32,
}

fn main() {
    let p = Persona { nombre: "Ana".into(), edad: 30 };
    println!("{:?}", p);
}
```

Rust

Útil para inspeccionar valores rápidamente.

1. Trait Clone

Permite clonar un valor: crear una copia profunda

```
#[derive(Clone)]
struct Punto {
    x: i32,
    y: i32,
}

let a = Punto { x: 1, y: 2 };
let b = a.clone();
```

Rust

`clone()` crea una copia real del contenido.

2. Trait Copy

Una extensión de Clone para tipos livianos y triviales de copiar.

Ejemplos:

- enteros
- booleanos
- caracteres
- flotantes
- tuplas pequeñas con solo tipos Copy


```
#[derive(Copy, Clone)]
struct Coordenada(i32, i32);

let a = Coordenada(1,2);
let b = a; // copia automática
```

Rust

3. Trait PartialEq

Permite comparar valores con == y !=.

```
#[derive(PartialEq)]
struct Usuario {
    id: u32,
}

let a = Usuario { id: 1 };
let b = Usuario { id: 1 };

assert!(a == b);
```

Rust

4. Trait Eq

Indica que la igualdad es “total”:

Casi siempre aparece junto a PartialEq.

```
#[derive(PartialEq, Eq)]
struct Dni(u32);
```

Rust

5. Trait PartialOrd y Ord

Permite usar <, >, <=, >= y ordenar colecciones.

```
#[derive(PartialOrd, Ord, PartialEq, Eq)]
struct Numero(i32);

let a = Numero(3);
let b = Numero(5);

assert!(a < b);
```

Rust

Ord permite ordenamiento total (necesario para sort()).

Definición básica de un Trait

```
trait Saludo {
    fn saludar(&self) -> String;
}
```

Rust

Aquí no hay implementación, solo se define el requisito:

Si un tipo implementa `Saludo`, debe tener el método `saludar`.

Implementando un Trait en un Struct

```
struct Persona {
    nombre: String,
}

impl Saludo for Persona {
    fn saludar(&self) -> String {
        format!("Hola, soy {}", self.nombre)
    }
}
```

Rust

uso:

```
let p = Persona { nombre: "Luis".into() };
println!("{}", p.saludar());
```

Rust

Un Trait como Interface de Comportamiento

Puedes tener muchos structs diferentes con el mismo trait:

```
struct Robot {
    id: u32,
}

impl Saludo for Robot {
    fn saludar(&self) -> String {
        format!("Unidad {} reportándose", self.id)
    }
}
```

Rust

Así tienes polimorfismo:

```
fn presentar(s: &impl Saludo) {
    println!("{}", s.saludar());
}

presentar(&Persona { nombre: "Ana".into() });
presentar(&Robot { id: 42 });
```

Rust

Sintaxis alternativa de generics para Traits

Estas dos formas son equivalentes:

1. Forma corta (impl Trait)

```
fn presentar(s: &impl Saludo) { ... }
```

Rust

2. Forma larga (generics + trait bound)

```
fn presentar<T: Saludo>(s: &T) { ... }
```

Rust

Las dos hacen lo mismo:

permiten que la función acepte cualquier tipo que implemente el trait.

Traits con Métodos por Defecto

Un trait puede contener implementaciones por defecto:

```
trait Animal {  
    fn sonido(&self) -> &str;  
  
    fn hablar(&self) {  
        println!("El animal hace: {}", self.sonido());  
    }  
}
```

Rust

Si tu struct implementa Animal pero no hablar, la versión por defecto se usará.

Traits con métodos estáticos

Los traits pueden tener métodos que no reciben self:

```
trait Identificable {  
    fn tipo() -> &'static str;  
}
```

Rust

Implementar un Trait genérico

Puedes usar generics dentro de los traits:

```
trait Sumar<T> {  
    fn sumar(&self, otro: T) -> T;  
}
```

Rust

Esto permite traits más flexibles.

Traits en Enums

Sí, los enums pueden implementar traits igual que los structs:

```
enum Estado {  
    Activo,  
    Inactivo,  
}  
  
impl Saludo for Estado {  
    fn saludar(&self) -> String {  
        match self {  
            Estado::Activo => "Estoy activo".into(),  
            Estado::Inactivo => "Estoy inactivo".into(),  
        }  
    }  
}
```

Rust

Supertraits

Un trait puede depender de otro:

```
trait A: B { }
```

Rust

Trait Bounds

Una restricción que indica que un tipo genérico debe implementar un trait específico.

Sin bounds, un genérico no puede usar ninguna funcionalidad del trait.

Sintaxis básica:

```
fn mostrar<T: Debug>(x: T) {  
    println!("{:?}", x);  
}
```

Rust

Múltiples bounds

```
fn procesar<T: Clone + PartialEq>(x: T, y: T) -> bool {  
    x == y  
}
```

Rust

Sintaxis alternativa con where

Útil cuando hay varios bounds:

```
fn combinar<T, U>(t: T, u: U)  
where  
    T: Debug,  
    U: Clone + Debug,  
{  
    println!("{:?}", t);  
}
```

Rust

Bounds en structs

```
struct Punto<T: Copy> {  
    x: T,  
    y: T,  
}
```

Rust

Bounds en impl

```
impl<T: PartialOrd> Maximo<T> {  
    fn max(a: T, b: T) -> T {  
        if a > b { a } else { b }  
    }  
}
```

Rust

Bounds en enums

```
enum Contenedor<T: Debug> {  
    Valor(T),  
}
```

Rust

Bounds súper-traits Un trait que requiere otro trait:

```
trait Imprimible: Display {  
    fn imprimir(&self) {  
        println!("{}", self);  
    }  
}
```

Rust

Lifetimes

Un lifetime es una etiqueta que describe cuánto tiempo vive una referencia.

No es un valor, no se almacena en memoria, no existe en tiempo de ejecución.

Solo es información estática para que el compilador asegure que no haya referencias colgantes.

Ejemplo de referencia peligrosa:

```
let r;  
  
{  
    let x = 10;  
    r = &x;    // x muere aquí  
}  
  
println!("{}", r); // r apunta a algo destruido
```

Rust

El compilador detecta este tipo de errores antes de que el programa se ejecute.

Notación ‘a

Una lifetime se escribe con un apóstrofo:

```
&'a i32
```

Rust

Se lee:

- “referencia a un i32 que vive al menos tanto como ‘a”.

¿Por qué las funciones necesitan lifetimes explícitas?

Cuando una función recibe y devuelve referencias, Rust no sabe por sí solo qué referencia vive más.

Ejemplo:

```
fn mayor(a: &str, b: &str) -> &str {  
    if a.len() > b.len() { a } else { b }  
}
```

Rust

Esto produce error:

“la referencia devuelta necesita un lifetime, pero no está claro cuál”.

Para solucionarlo:

```
fn mayor<'a>(a: &'a str, b: &'a str) -> &'a str {  
    if a.len() > b.len() { a } else { b }  
}
```

Rust

Significado:

- ambos parámetros viven al menos 'a
- el retorno también vive 'a
- por lo tanto, el retorno es válido

Ejemplo completo de Lifetimes en funciones

```
fn el_mas_largo<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

Rust

uso:

```
let s1 = String::from("hola");  
let s2 = String::from("rustaceo");  
  
let res = el_mas_largo(s1.as_str(), s2.as_str());  
  
println!("{}", res);
```

Rust

Lifetimes en Structs

Cuando un struct contiene referencias, debe tener lifetimes explícitas.

```
struct Texto<'a> {  
    contenido: &'a str,  
}  
  
let s = String::from("Hola");  
let t = Texto { contenido: &s };
```

Rust

El struct vive menos o igual que 'a.

Lifetimes con métodos

```
struct Contenedor<'a> {  
    valor: &'a str,  
}  
  
impl<'a> Contenedor<'a> {  
    fn obtener(&self) -> &'a str {  
        self.valor  
    }  
}
```

Rust

El 'a del método coincide con el 'a del struct.

Lifetimes en Enum

```
enum Mensaje<'a> {  
    Texto(&'a str),  
    Numero(i32),  
}
```

Rust

Funcionan igual que en estructuras.

Reglas de Lifetime Elision

Rust aplica estas reglas cuando puede inferir sin ambigüedad:

Regla 1

Cada parámetro de entrada obtiene su propio lifetime.

```
fn f(x: &i32, y: &i32);
```

Rust

Se convierte en:

```
fn f<'a, 'b>(x: &'a i32, y: &'b i32);
```

Rust

Regla 2

Si hay un solo parámetro de entrada, el retorno recibe su lifetime.

```
fn id(x: &i32) -> &i32; // lifetime elision
```

Rust

Equivalente a:

```
fn id<'a>(x: &'a i32) -> &'a i32;
```

Rust

Regla 3

Si el método tiene &self o &mut self, el retorno hereda ese lifetime.

```
fn valor(&self) -> &str;
```

Rust

Equivalente a:

```
fn valor<'a>(&'a self) -> &'a str;
```

Rust

Lifetime Bounds Puedes usar lifetimes como restricciones:

```
fn comparar<'a, T: PartialEq + 'a>(x: &'a T, y: &'a T) -> bool {  
    x == y  
}
```

Rust

T: 'a significa:

- “T vive al menos tanto como ‘a”.

Múltiples lifetimes

Puede haber varias lifetimes independientes:

```
fn elegir<'a, 'b>(a: &'a str, b: &'b str) -> &'b str {  
    b
```

Rust

```
}
```

Significa:

- ‘a y ‘b no dependen uno del otro
- el retorno depende de ‘b

Static lifetime ‘static

Es el lifetime más largo posible.

Significa:

- vive toda la duración del programa

```
let x: &'static str = "cadena literal";
```

Rust

Los literales string siempre son ‘static.

Ejemplo Real: dividir texto en partes

```
fn primera_palabra<'a>(s: &'a str) -> &'a str {
    let bytes = s.as_bytes();

    for (i, &c) in bytes.iter().enumerate() {
        if c == b' ' {
            return &s[..i];
        }
    }

    s
}
```

Rust

Este patrón aparece muchísimo:

función lee referencia y devuelve una subreferencia válida → lifetime ‘a.

Lifetimes y Traits juntos

Puedes tener traits parametrizados con lifetimes:

```
trait Logger<'a> {
    fn log(&self, msg: &'a str);
}
```

Rust

O implementar traits para tipos con lifetimes:

```
impl<'a> Default for Texto<'a> {
    fn default() -> Self {
        Self { contenido: "" }
    }
}
```

Rust

Bounds con lifetimes


```
fn mostrar<'a, T>(x: &'a T)
where
    T: Debug + 'a,
{
    println!("{:?}", x);
}
```

Rust

Manejo de Errores

Errores Recuperables

Rust trata el manejo de errores como una parte esencial del lenguaje.

Para evitar fallos inesperados, Rust propone una estrategia distinta a otros lenguajes: tratar los errores como valores, no como excepciones.

¿Qué es un error recuperable?

Un error recuperable es aquel que:

- Podemos anticipar
- Podemos manejar
- No requiere terminar el programa

Ejemplos:

- Un archivo que no existe
- Una conexión de red que falla
- Un número que no se puede convertir
- Un usuario que podría o no estar registrado

Rust usa `Option` y `Result` para estos casos.

`Option<T>`

valor o ausencia de valor

```
enum Option<T> {
    Some(T),
    None,
}
```

Rust

Sirve para modelar cosas que pueden o no existir.

Obtener el primer elemento de un vector

```
fn obtener_primer(nums: &Vec<i32>) -> Option<i32> {
    nums.first().copied()
}
```

Rust

Uso:

```
match obtener_primer(&vec![10, 20, 30]) {
    Some(n) => println!("Primer número: {}", n),
}
```

Rust

```
None => println!("El vector está vacío"),
}
```

Métodos importantes de `Option<T>`

- `.unwrap()`
Obtiene el valor o hace panic
- `.unwrap_or(x)`
Valor por defecto
- `.unwrap_or_else()`
Llama una función para obtener un valor
- `.map()`
Transforma `Some`
- `.and_then()`
Encadena operaciones que también devuelven `Option`

Ejemplo: evitar panic usando `unwrap_or`

```
let nombre = Some("Juan");
let n = nombre.unwrap_or("Invitado");
```

Rust

Result<T, E>

operación que puede fallar

Definición:

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

Rust

Ejemplos: abrir archivos, parsear strings, convertir tipos.

Convertir un string en número

```
fn convertir(s: &str) -> Result<i32, std::num::ParseIntError> {
    s.parse::<i32>()
}
```

Rust

Uso:

```
match convertir("42") {
    Ok(n) => println!("Número: {}", n),
    Err(e) => println!("Error: {}", e),
}
```

Rust

Métodos importantes de `Result<T, E>`

- `.unwrap()`
Obtiene `T` o panic
- `.expect(msg)`

Igual que unwrap pero con mensaje

- `.map()`

Transforma Ok

- `.map_err()`

Transforma Err

- `.ok()`

Convierte Result a Option

- `.err()`

Obtiene la parte Err

El operador ?

El operador ?:

- Si es Ok, devuelve el valor
- Si es Err, retorna inmediatamente de la función

Ejemplo sin ?

```
fn leer_archivo() -> Result<String, std::io::Error> {
    let mut archivo = File::open("hola.txt");
    match archivo {
        Ok(mut f) => {
            let mut contenido = String::new();
            match f.read_to_string(&mut contenido) {
                Ok(_) => Ok(contenido),
                Err(e) => Err(e),
            }
        }
        Err(e) => Err(e),
    }
}
```

Rust

Ejemplo con ?

```
fn leer_archivo() -> Result<String, std::io::Error> {
    let mut contenido = String::new();
    File::open("hola.txt")?.read_to_string(&mut contenido)?;
    Ok(contenido)
}
```

Rust

Patrones para manejar errores correctamente

match

```
match archivo {
    Ok(a) => println!("Abierto"),
    Err(e) => println!("Error: {}", e),
}
```

Rust

if let

```
if let Some(n) = x {  
    println!("El número es {}", n);  
}
```

Rust

while let

Ideal para iteradores:

```
let mut iter = vec![1,2,3].into_iter();  
while let Some(v) = iter.next() {  
    println!("{}", v);  
}
```

Rust

Errores NO Recuperables

¿Qué es un error no recuperable?

Un error no recuperable es cuando la única opción segura es detener el programa inmediatamente para evitar corrupción de datos o comportamiento impredecible.

Ejemplos:

- Índice fuera de rango en un vector
- Dividir entre cero
- Usar unwrap() sobre None o Err
- Violaciones lógicas del código (invariantes imposibles)

Rust usa panic! para estos casos.

panic!

Detener el programa de forma controlada

panic! es una macro que termina la ejecución y muestra un mensaje.

```
fn main() {  
    panic!("Ocurrió un error inesperado");  
}
```

Rust

Salida típica:

```
thread 'main' panicked at 'Ocurrió un error inesperado', src/main.rs:2:5
```

Rust

¿Cuándo ocurre un panic automáticamente?

1. Índices fuera de rango

```
let v = vec![1, 2, 3];  
println!("{}", v[10]); // panic!
```

Rust

1. unwrap() sobre None

```
let x: Option<i32> = None;  
x.unwrap(); // panic
```

Rust

1. `expect()` también provoca panic

```
let r: Result<i32, _> = "hola".parse();  
r.expect("No se pudo convertir"); // panic con mensaje personalizado
```

Rust

Cuándo no usar `unwrap()` y `expect()`

Ambos métodos causan panic si:

- `Option` es `None`
- `Result` es `Err`

¿Por qué existen entonces?

Porque en prototipos, scripts rápidos y tests es conveniente usarlos.

Cuándo evitar `unwrap/expect`:

- En producción
- En código que maneja errores del usuario
- En operaciones de I/O, red, archivos
- Cuando el valor puede no existir o fallar

Backtrace

seguir el origen del panic

Rust puede mostrar la ruta de llamadas que llevó al error.

Para activarlo:

```
RUST_BACKTRACE=1 cargo run
```

Rust

Ejemplo de salida:

```
thread 'main' panicked at 'fallo X', src/main.rs:12:9  
stack backtrace:  
 0: rust_begin_unwind  
 1: core::panicking::panic_fmt  
 2: my_project::funcion  
 3: my_project::main
```

Rust

Esto es útil para depurar.