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

Primeros pasos

¿Qué es Rust?

Rust es un lenguaje de sistemas moderno, diseñado para garantizar seguridad de memoria sin recolector de basura y rendimiento comparable a C y C++.

Su desarrollo comenzó en 2006 por Graydon Hoare y fue patrocinado por Mozilla, culminando con su primera versión estable en mayo de 2015.

Rust resuelve problemas clásicos de lenguajes de bajo nivel como punteros nulos, doble liberación y condiciones de carrera **data races**, mediante un sistema de propiedad **Ownership** y un verificador de préstamos **Borrowing** que el compilador valida en tiempo de compilación.

Rust no solo permite crear aplicaciones completas desde cero, puesto que su diseño de abstractions y memory safety lo convierte en una opción atractiva para reescribir componentes críticos en ecosistemas ya existentes.

Ejemplos concretos:

1. Firefox sustituyó partes de su motor CSS con Servo, reduciendo errores de memoria en un 70 %.
2. Discord migró su servicio de voz a Rust y dividió por 10 el uso de CPU frente a la versión en Go.
3. Dropbox reescribió su sistema de sincronización con Rust + Tokio, logrando menos latencia y menor consumo de memoria sin aumentar el coste de hardware.

Estas migraciones demuestran que Rust puede aumentar el rendimiento y reducir costes operativos sin sacrificar seguridad ni productividad.

La seguridad garantizada por su diseño ha generado una confianza excepcional entre los desarrolladores. De manera ininterrumpida desde 2016, Rust ha sido votado como el “lenguaje de programación más querido” en la encuesta anual de Stack Overflow.

Esta popularidad y la alta satisfacción de los desarrolladores son un fuerte indicador de su productividad y baja curva de frustración una vez dominado.

El compilador de Rust puede parecer exigente al principio, pero está diseñado para detectar errores *antes* de que tu código se ejecute. Esto significa menos bugs en producción y noches más tranquilas.

Además, Rust:

- **Compila directamente a código máquina**: Sin overhead de runtime ni máquinas virtuales
- **Soporta múltiples hilos sin data races**: El compilador garantiza que tu código concurrente es seguro
- **Elimina categorías enteras de bugs**: Muchos errores de concurrencia simplemente no pueden ocurrir

Por qué las empresas eligen Rust

Las razones principales que impulsan esta adopción son:

1. **Menos bugs en producción**: El compilador atrapa errores antes del despliegue
2. **Mayor rendimiento**: Comparable a C/C++ sin sacrificar seguridad
3. **Menor uso de recursos**: Traduce en ahorro de costos de infraestructura
4. **Código más mantenible**: Las garantías del lenguaje facilitan refactorings grandes

Instalación de Rust

Rust se instala mediante rustup, el instalador oficial que gestiona versiones del compilador, toolchains y herramientas asociadas. Este proceso es prácticamente idéntico en todos los sistemas operativos.

Linux y macOS

1. Abre una terminal.
2. Ejecuta el siguiente comando para descargar e iniciar el instalador de Rust:

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

Este comando descarga el instalador de **rustup**, que configurará:

1. El compilador Rust **rustup**
2. **Cargo**, el gestor de paquetes y herramientas de construcción
3. La documentación estándar
4. Las herramientas necesarias para trabajar con Rust

Durante el proceso, se te preguntará si deseas continuar con la instalación por defecto o personalizarla.

La opción recomendada es presionar Enter para aceptar la configuración estándar.

Verás algo como esto:

```
--snip--
Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, you need to source
the corresponding env file under $HOME/.cargo.
```
Terminal

Reinicia tu terminal o ejecuta:

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

Windows

Rust en Windows requiere el compilador **MSVC** de Microsoft para generar archivos ejecutables **.exe**. Debes instalar **Visual Studio Build Tools** antes de instalar Rust.

1. Instalar Visual Studio Build Tools

   Descarga Visual Studio desde:

   https://visualstudio.microsoft.com/es/downloads/

2. Componentes requeridos

   Rust necesita el workload **Desktop Development with C++** que incluye todas las herramientas esenciales para compilar programas nativos.

3. Instalación minimalista

Si no planeas desarrollar en C++ y quieres instalar solo lo indispensable:

Ve a la pestaña **Componentes individuales** y selecciona unicamente:

- **MSVC v143 – VS 2022 C++ x64/x86 build tools**
- **Windows 11 SDK (10.0.22621.0)**

Después de seleccionar los componentes, haz clic en Instalar y espera a que finalice.

4. Descargar Rust desde la página oficial

   Accede a la página oficial:

   https://www.rust-lang.org/tools/install

   Aquí encontrarás tres instaladores según la arquitectura de tu sistema:

   - rustup-init.exe (32 bits)
   - rustup-init.exe (x64 / 64 bits)
   - rustup-init.exe (ARM64)

   Selecciona el que corresponda a la arquitectura de tu sistema.

5. Ejecutar el instalador

   Ejecuta el archivo **rustup-init.exe** y sigue las instrucciones de la consola.

   Cuando aparezca el menú, simplemente presiona Enter para la instalación estándar:

   ```
   --snip--                                                    Terminal
   1) Proceed with standard installation (default - just press enter)
   2) Customize installation
   3) Cancel installation
   >1
   ```

   En la mayoría de casos es necesario cerrar y volver a abrir la terminal para que Windows reconozca los nuevos comandos **rustc cargo rustup** y sus todas sus herramientas.

6. Verificación de la Instalación

   Ejecuta los siguientes comandos en tu terminal:

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

7. Solución de Problemas Comunes

   1. Linux/macOS: “comando no encontrado”

      Si después de instalar Rust sigues viendo command not found:

      ```
      # Verifica que $HOME/.cargo/bin esté en tu PATH             Terminal
      echo $PATH | grep .cargo

      # Si no aparece, añádelo manualmente
      echo 'export PATH="$HOME/.cargo/bin:$PATH"' >> ~/.bashrc
      ```

```
source ~/.bashrc
```

)

1. Windows: “rustc no se reconoce como comando”
   - Reinicia tu terminal. Este es el problema más común.
   - Verifica que la instalación se completó sin errores.
   - Verifica manualmente que existe:

```
dir $env:USERPROFILE\.cargo\bin\rustc.exe
```
Terminal

Gestión de Versiones con Rustup

1. Actualizar Rust

```
rustup update
```
Terminal

1. Cambiar entre canales

```
rustup default stable   # Canal estable (recomendado)
rustup default beta     # Canal beta
rustup default nightly  # Canal nightly (experimental)
```
Terminal

No necesitas instalar **nightly** a menos que uses características experimentales o herramientas específicas que lo requieran.

Cargo

Cargo es el sistema de compilación y gestor de paquetes oficial de Rust. Es una herramienta fundamental que simplifica todo el ciclo de desarrollo, desde la creación de proyectos hasta la gestión de **dependencias** y la **compilación**.

Cargo se instala automáticamente cuando instalas Rust a través de rustup.

- Crea y estructura proyectos
- Compila tu código
- Descarga y gestiona dependencias
- Ejecuta tests
- Genera documentación
- Publica paquetes en crates.io

Compilación y Ejecución en Rust

La ejecución de código en Rust puede realizarse mediante dos vías fundamentales:
- Compilador directo, rustc, para tareas sencillas.
- Cargo la herramienta estándar de gestión de proyectos, indispensable para el desarrollo moderno.

Usando rustc

rustc es el compilador oficial de Rust. Es el programa que transforma tu código fuente (archivos .rs) en ejecutables que tu computadora puede ejecutar.

Entender **rustc** te ayuda a comprender mejor lo que sucede “bajo el capó”.

Imagina que escribes una receta en español, pero tu horno solo entiende instrucciones en lenguaje de máquina. El compilador **rustc** es el traductor que convierte tu receta **código Rust** en instrucciones que el horno **CPU** puede ejecutar.

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.

Fases del Proceso

1. Paso 1: Creación del Módulo Fuente

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

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

   - A:

     Define la función principal y obligatoria de un programa ejecutable en Rust.

   - B:

     Es un nombre reservado y especial que el compilador de Rust y el sistema operativo buscan para saber dónde empezar a ejecutar el código.

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
   ```

   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
     ```

   - Linux/macOS:

     ```
     ./main
     ```

   Resultado:

   ```
   Compilador directo rustc.
   ```

Tu Primer Proyecto con Cargo

Crear un nuevo proyecto

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

```
cargo new a hola_mundo b
```

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

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

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

```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
```
toml

(1) Metadatos

(2) Librerías externas

Punto de entrada: main.rs

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

```
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
```
Terminal

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!
```
Terminal

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

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`.
```
Terminal

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**.

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

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.

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

- 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".

```rust
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 { }.

```rust
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.

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

fn main() {
  const PI: f64 = 5.14; C
```
Rust

```
  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).

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

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

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.

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

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

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

     ```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*
     ```

     Python

     ```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
     ```

   - 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 devuleve un unit type ().

```rust
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.

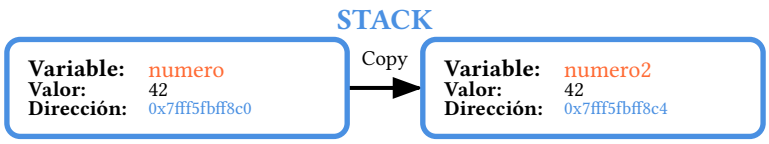
Los primeros conceptos pilares es comprender como se administra la memoria. Rust divide la memoria en dos grandes regiones Stack y Heap, su diferencia es esencial para evitar errores como:

- 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.

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



Los valores se copian porque `i32` 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.

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

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

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 razon 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 invalido, ya que fue movida a texto2.

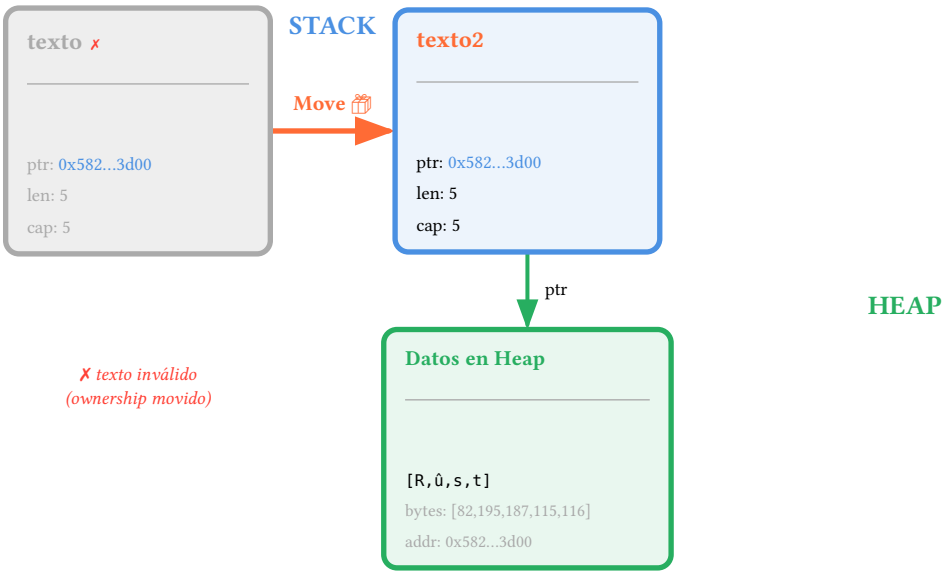


Diagrama 3: Move semántico: el ownership se transfiere de `texto` a `texto2`

Tipos de datos

En Rust, un tipo de dato define la naturaleza de la información que una variable puede almacenar y las operaciones que se pueden realizar sobre dicha información. El sistema de tipos de Rust es estático y fuertemente tipado, lo que significa que el compilador conoce el tipo de cada valor en tiempo de compilación y no permite operaciones entre tipos incompatibles sin una conversión explícita.

Por ejemplo, una variable declarada como `i32` solo puede almacenar números enteros de 32 bits con signo, y no puede ser utilizada directamente como una cadena de texto o un número de punto flotante:

```rust
let numero: i32 = 10;
// let texto: String = numero; // Error: tipos incompatibles
```

Importancia del sistema de tipos en Rust

El sistema de tipos de Rust es el eje central que sustenta la seguridad, el rendimiento y la confiabilidad del lenguaje. Gracias a verificaciones exhaustivas en tiempo de compilación, Rust previene errores comunes como incompatibilidades de tipos, accesos inválidos a memoria y condiciones de carrera, evitando fallos en tiempo de ejecución. Este sistema se integra con los conceptos de ownership, borrowing y lifetimes para garantizar seguridad de memoria sin necesidad de recolector de basura. Además, el conocimiento completo de los tipos permite al compilador generar código altamente optimizado, logrando un rendimiento comparable a C y C++. Finalmente, los tipos aportan claridad y mantenibilidad al código, funcionando como documentación implícita y facilitando la evolución de proyectos complejos.

Tipos de Datos en Rust

En el ecosistema de Rust, todo valor pertenece a un tipo de dato específico. Estos se dividen en dos grandes categorías según cómo organizan la información en la memoria: tipos escalares y tipos compuestos.

Scalar Types

Representan un único valor. En Rust, los principales tipos escalares son los enteros, los números de punto flotante, el tipo booleano y el tipo carácter. Estos tipos son fundamentales y suelen almacenarse directamente en el stack, lo que permite un acceso rápido y eficiente.

Enteros

Los enteros son tipos de datos numéricos que representan valores sin componente fraccionaria. En Rust, los tipos enteros 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 su capacidad para representar números negativos:

Enteros con Signo (`i`)

Los enteros con signo pueden representar valores positivos, negativos y cero. Utilizan el sistema de complemento a dos para la representación de números negativos, donde el bit más significativo indica el signo del número.

Tipo	Valor mínimo	Valor máximo
`i8`	−128	127
`i16`	−32 768	32 767
`i32`	−2 147 483 648	2 147 483 647
`i64`	−9 223 372 036 854 775 808	9 223 372 036 854 775 807
`i128`	-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.

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

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
u8	0	255
u16	0	65 535
u32	0	4 294 967 295
u64	0	18 446 744 073 709 551 615
u128	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.

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

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
usize	0	$2^{32} - 1$ o $2^{64} - 1$
isize	-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

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

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.

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

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

Sufijos de Tipo en Literales

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

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

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:

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

Bases Numéricas

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

```rust
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)
```

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

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

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.

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

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.

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

Control Explícito del Desbordamiento

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

```rust
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)
```

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`:

```rust
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)
```

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:

```rust
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);
```

Operaciones y métodos comunes

```rust
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());
```

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
`+=`	`x = x + y`	`x += 2;`
`-=`	`x = x - y`	`x -= 3;`
`*=`	`x = x * y`	`x *= 5;`
`/=`	`x = x / y`	`x /= 2;`
`%=`	`x = x % y`	`x %= 3;`

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.

```rust
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:

```rust
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:

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

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:

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

Valores Especiales

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

```rust
// 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
```

Verificación de Valores Especiales

Rust proporciona métodos para detectar estos casos:

```rust
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());
```

Constantes y Límites

Todos los tipos de punto flotante proporcionan constantes útiles:

```rust
// Límites numéricos
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

```rust
let valor = 3.7;

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

```rust
let negativo = -15.8;
```

```
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

```rust
let base = 4.0;

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

```rust
let numero:f32 = 42.75;

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:

```rust
let valor = 123.456789;

// 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:

```rust
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;
```

```rust
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
```

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

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

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: `true` se convierte en `false` y viceversa
&&	AND : Conjunción	Devuelve `true` solo si ambos operandos son `true`
||	OR : Disyunción	Devuelve `true` si al menos uno de los operandos es `true`
^	XOR : Disyunción exclusiva	Devuelve `true` solo si exactamente uno de los operandos es `true`

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
`!true`	`false`
`!false`	`true`

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
true	true	true
true	false	false
false	true	false
false	false	false

Tabla 9: Tabla de verdad del operador AND

Ejemplo práctico:

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

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

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:

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

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

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

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

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`:

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

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

Precedencia de Operadores

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

1. `!`: Mayor precedencia
2. Operadores de comparación (==, !=, <, >, <=, >=)
3. &&
4. ||
5. ^: Menor precedencia

```rust
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
```

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 (`"`):

```rust
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';
```

Métodos de Inspección

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

Clasificación de Caracteres

```rust
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

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

Tabla 13: Secuencias de escape para caracteres

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

```rust
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);
```

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:

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

Rust también puede inferir los tipos:

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

Acceso por índice

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

Los índices empiezan en 0.

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

Desestructuración

Desempaqueta la tupla en variables:

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

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

Ignorar valores con '_'

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

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

Debug en tuplas

Puedes imprimir una tupla con:

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

Siempre que todos sus elementos implementen `Debug`.

Mutabilidad

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

Ejemplo:

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

```
persona.2 = false;

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

Resultado:

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

Tupla vacía

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

También se llama:
- unit type
- unit value

```rust
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:

```rust
let tupla = (1, 2, 3, 4, 5);
let (primero, .., ultimo) = tupla;

println!("primero = {}, ultimo = {}", primero, ultimo);
// primero = 1, ultimo = 5
```
Rust

```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:

```rust
let anidada = ((1, 2), (3, 4));

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

Para mayor claridad, puedes desestructurar:

```rust
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
```

Tuplas con un solo elemento

Esto confunde a muchos:

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

Las tuplas de un solo elemento deben llevar coma final.

Comparaciones

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

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

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

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**.

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

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

```rust
// 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
```

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

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

Creación

Se definen entre corchetes '`[ ]`'.

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

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

Resultado con `{:#?}`:

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

Tipos de datos

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

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

Si mezclas tipos distintos, Rust no compila:

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

Acceso por índice

```rust
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
```

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.

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

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

Acceder fuera del rango genera un panic!.

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

```Rust
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
```

Declaración de arrays

Inicialización explícita

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

Inicialización repetida

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

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

Útil para inicializar arrays grandes:

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

Mutabilidad

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

```Rust
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]
```

No puedes cambiar el tamaño, solo los valores:

```rust
let mut arr = [1, 2, 3];
// arr[3] = 4;  // Error: índice fuera de rango
```

Desestructuración

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

// 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:

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

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

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

Rest pattern

Puedes tomar los primeros elementos y manejar el resto:

```rust
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);
```

Resultado:

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

También puedes capturar desde el final:

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

```rust
let [.., penultimo, ultimo] = nums;

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

Comparaciones

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

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

```rust
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

```rust
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

```rust
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:

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

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

Métodos útiles

```rust
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)
```

Resultado:

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

Métodos adicionales útiles

```rust
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]
```

Conversión entre tuplas y arrays

De tupla a array

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

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

Esto solo funciona para tuplas con elementos del mismo tipo.

De array a tupla

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

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

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

```rust
// ✓ 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];
```

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![]`

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

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

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

3. Vector con elementos repetidos

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

4. Vector a partir de otro vector

```rust
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]
```

5. Vector a partir de un iterador

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

6. Vector con capacidad reservada

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

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

7. Vector a partir de un array

```rust
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]
```

Acceso a elementos

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

1. Usando índice directo `[]`

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

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

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

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

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

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

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

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

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

3. Primera y última posición

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

println!("{:?}", v.first());  // Some(10)
println!("{:?}", v.last());   // Some(40)
```

Ambos retornan `Option<&T>`.

Mutabilidad

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

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

v[1] = 50;  // cambia 20 => 50

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

También puedes obtener referencias mutables:

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

if let Some(primer) = v.first_mut() {
    *primer = 100;
}

println!("{:?}", v);  // [100, 2, 3]
```

Agregar elementos

1. Agregar al final con `push()`

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

v.push(3);
v.push(4);

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

2. Agregar múltiples elementos con `extend()`

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

```
v.extend([4, 5, 6]);

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

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

```rust
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)
```

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

4. Cambiar tamaño con `resize()`

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

v.resize(3, "mundo");

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

Si el nuevo tamaño es menor, trunca el vector:

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

v.resize(2, 0);

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

Insertar elementos

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

```rust
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]
```

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

Eliminar elementos

1. Eliminar por índice con `remove()`

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

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

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

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

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

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

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

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

```rust
let mut v: Vec<i32> = vec![];
println!("{:?}", v.pop());  // None
```

3. Eliminar un rango con `drain()`

```rust
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]
```

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

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

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

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

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

La capacidad se mantiene:

```rust
let mut v = vec![1, 2, 3, 4, 5];
println!("Capacidad antes: {}", v.capacity());

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

5. Acortar con `truncate()`

Mantiene los primeros n elementos y elimina el resto:

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

v.truncate(2);

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

Para vaciar todo:

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

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):

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

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()`

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

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

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

8. Eliminar duplicados consecutivos con `dedup()`

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

v.dedup();

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

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

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

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

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

Intercambio y manipulación

1. Intercambiar elementos con `swap()`

```Rust
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"]
```

2. Invertir el vector con `reverse()`

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

v.reverse();

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

3. Rotar elementos

Rotar a la izquierda:

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

v.rotate_left(2);

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

Rotar a la derecha:

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

v.rotate_right(2);

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

4. Dividir el vector con `split_off()`

```Rust
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]
```

Longitud y capacidad

Rust distingue entre:

- `len()` ⟶ cantidad actual de elementos

- `capacity()` ⟶ espacio reservado en memoria

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

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:

```rust
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)
```

Métodos relacionados con capacidad:

```rust
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
```

Copia y movimiento

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

Al asignarlos, se mueven, no se copian automáticamente:

```rust
let v1 = vec![1, 2, 3];
let v2 = v1;  // v1 se mueve a v2
```

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

Para mantener ambos, usa `clone()`:

```rust
let v1 = vec![1, 2, 3];
let v2 = v1.clone();  // copia explícita

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

Comparaciones

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

```rust
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)
```

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

Ordenamiento

1. Ordenar con `sort()`

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

v.sort();

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

2. Ordenar de mayor a menor

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

v.sort_by(|a, b| b.cmp(a));

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

3. Ordenar con función personalizada `sort_by_key()`

```rust
let mut personas = vec![
    ("Ana", 25),
    ("Luis", 30),
    ("Pedro", 20),
];
```

```rust
personas.sort_by_key(|persona| persona.1);  // ordenar por edad

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

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

```rust
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]
```

Búsqueda

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

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

println!("{}", v.contains(&20));  // true
println!("{}", v.contains(&99));  // false
```

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

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

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

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

```rust
let v = vec![1, 3, 5, 7, 9];

match v.binary_search(&5) {
    Ok(pos) => println!("Encontrado en: {}", pos),  // 2
    Err(_) => println!("No encontrado"),
}
```

Vectores multidimensionales

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

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

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

Crear una matriz de tamaño dinámico:

```rust
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]]
```

Iteración

1. Iterar por valor (consume el vector)

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

for num in v {
    println!("{}", num);
}

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

2. Iterar por referencia inmutable

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

for num in &v {
    println!("{}", num);
}

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

3. Iterar por referencia mutable

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

for num in &mut v {
    *num *= 2;
}

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

4. Iterar con índice usando `enumerate()`

```rust
let v = vec!["a", "b", "c"];
```

```rust
for (i, valor) in v.iter().enumerate() {
    println!("v[{}] = {}", i, valor);
}
```

Métodos de concatenación

1. Concatenar slices con `concat()`

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

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

2. Unir con separador usando `join()`

```rust
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
```

Métodos de verificación

```rust
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
```

Repetir vector

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

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

Conversión entre tipos

De vector a array (tamaño conocido)

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

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

De vector a slice

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

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

Cuándo usar Vec vs Array

Usa `Vec<T>` cuando:
- El tamaño no se conoce en tiempo de compilación
- Necesitas agregar o eliminar elementos dinámicamente
- Trabajas con colecciones grandes (que no caben en el stack)
- Necesitas flexibilidad sobre rendimiento

Usa `[T; N]` cuando:
- El tamaño es fijo y conocido
- Quieres máxima velocidad (stack allocation)
- Trabajas con colecciones pequeñas (< 1000 elementos)
- No necesitas modificar el tamaño

Resumen rápido

```rust
// Creación
let v1 = vec![1, 2, 3];
let v2 = Vec::new();
let v3 = vec![0; 5];

// Agregar/Eliminar
v.push(4);
v.pop();
v.insert(1, 99);
v.remove(0);
v.clear();

// Acceso
let x = v[0];
let y = v.get(1);  // Option<&T>

// Capacidad
v.len();
v.capacity();
v.is_empty();
```

```
// Ordenar/Buscar
v.sort();
v.reverse();
v.contains(&5);

// Iteración
for x in &v { }
for x in &mut v { }
```