

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 tus necesidades.

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.

```
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
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.

```
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
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

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

Rust

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
```

Rust

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
```

Rust

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

Rust

Operaciones Matemáticas

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

Redondeo

```
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

```
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

```
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

```
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:

```
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: {:.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; Rust

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
```

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; Rust
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:

```
let edad = 25;
let tiene_licencia = true;
let puede_conducir = edad >= 18 && tiene_licencia;
println!("{}", puede_conducir); // true
```

Rust

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); // 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); // 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
==	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:

```
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:

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

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

Rust

```
// = 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 ("):

```
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

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

// Control y formato
let tab = '\t';
```

Rust

```
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() {
```

Rust


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

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];
// Índice: 0, 1, 2, 3
```

Rust

- `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];
```

Rust

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

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];

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

Rust

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) {
```

Rust

```
Some(valor) => println!("Valor: {}", valor),
None => println!("Índice fuera de rango"),
}
// Imprime: Índice fuera de rango
```

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];

// 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
```

Rust

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];
let b = [1, 2, 3];
let c = [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.

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]
```

Rust

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();
```

Rust

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
```

Rust

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
```

Rust

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];
```

Rust

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

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

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");
```

Rust

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

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);

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

Rust

Para vaciar todo:

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

v.truncate(0);

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

Rust

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];

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

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

Rust

8. Eliminar duplicados consecutivos con dedup()

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

v.dedup();

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

Rust

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

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

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

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

Rust

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);

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
```

Rust

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
```

Rust

```
v.push(3); // se queda sin espacio, realocar
println!("cap: {}", v.capacity()); // 4 (duplicado)
```

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
```

Rust

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

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

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

Rust

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
```

Rust


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

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

Rust

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();
```

Rust

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

2. Unir con separador usando join()

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

Rust

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

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

Rust

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

Métodos de verificación

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

Rust

```
// 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

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

Rust

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

Conversión entre tipos

De vector a array (tamaño conocido)

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

Rust

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

De vector a slice

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

Rust

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

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

Rust

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
```

Rust

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

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
```

Rust

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

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";
```

Rust

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

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";
```

Rust


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

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

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";

for parte in s.rsplit(',') {
    println!("{}", parte);
}

// tres
// dos
// uno
```

Rust

Reemplazar contenido

1. Reemplazar todas las ocurrencias con replace()

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

let nuevo = s.replace("Hola", "Adiós");

println!("{}", nuevo); // Adiós Mundo Adiós
```

Rust

2. Reemplazar n ocurrencias con replacen()

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

let nuevo = s.replacen("Hola", "Hey", 2);

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

Rust

3. Reemplazar un rango

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

s.replace_range(5..10, "Rust");

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

Rust

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

Eliminar contenido

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

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

let ultimo = s.pop();

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

Rust

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");
```

Rust

```
s.truncate(4);

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

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();

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

Rust

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  ";
```

Rust

```
let limpio = s.trim_end();

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

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();

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

Rust

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";
```

Rust

```
let repetido = s.repeat(3);

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

```
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");
```

Rust

```
let s3 = String::from(" Rust");

let resultado = s1 + &s2 + &s3;

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

Rust

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

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

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,
    And so are you!
";

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

Rust

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";
```

Rust

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

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

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

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

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;
```

Rust


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

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";

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

Rust

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;

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

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

Rust

Siempre necesitas mut para modificar un HashMap.

2. Usando from() con array de tuplas

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

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

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

Rust

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"),
}
```

Rust

```
// Ana tiene 10 puntos
```

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

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

Rust

⚠ 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);
```

Rust

```

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

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

```

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

```

use std::collections::HashMap;

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

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

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

println!("¿Vacío? {}", 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;

```

Rust

```
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}
```

4. Actualizar basado en el valor anterior

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

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

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

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

let mut mapa = HashMap::new();

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

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

La función solo se ejecuta si la clave no existe.

Eliminar elementos

1. Eliminar con remove()

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

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}
```

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;

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
```

Rust

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

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

Rust

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

for valor in mapa.values() {
    println!("{}", valor);
}
```

Rust

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);
```

Rust

```
for (clave, valor) in mapa { // sin &, consume el mapa
    println!("{:}: {:?}", clave, valor);
}

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

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)
```

Rust

```
.or_insert(1);

println!("{:?}", mapa); // {"a": 15, "b": 1}
```

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}
// Solo imprime "Calculando..." una vez
```

Rust

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}
```

Rust

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);
```

Rust

```
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);
```

Rust

✗ 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)];

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]}
```

Rust

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;
```

Rust

```
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
```

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());

println!("clave: {}, valor: {}", clave, valor); // ✓ funcionan
```

Rust

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);
```

Rust

```

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

```

HashMap con structs personalizados

```

use std::collections::HashMap;

#[derive(Debug)]
struct Persona {
    nombre: String,
    edad: u32,
}

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)]

```

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();

println!("{:?}", claves);

```

Rust

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

```

Rust


```

for (clave, valor) in items {
    println!("{}", clave, valor);
}
// a: 1
// b: 2
// c: 3

```

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");

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);
    }
}

```

Rust

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 `HashMap`s 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
```

Rust

```
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);
```

Control flow