

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:

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

Rust

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.

```
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

Todos los elementos deben tener el mismo tipo exacto.

Si mezclas tipos distintos, Rust no compila:

```
// x 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];

// x Error en tiempo de ejecución (panic):
println!("{}", arr[5]);
// thread 'main' panicked at 'index out of bounds'
```

Rust

Acceder fuera del rango genera un panic (error en tiempo de ejecución).

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; // x 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

Usando 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]
```

makefile

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 (orden lexicográfico).

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; // x 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

// Iterar
for valor in arr {
    println!("{}", valor);
}

// Iterar con índice
for (i, valor) in arr.iter().enumerate() {
    println!("arr[{}] = {}", i, valor);
}

// 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 (manual)

```
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

Arrays vs Tuplas vs Vec

Usa arrays cuando:

- Necesitas una colección de tamaño fijo conocido en compilación
- Todos los elementos son del mismo tipo
- Quieres máximo rendimiento (stack allocation)
- Trabajas con colecciones pequeñas a medianas

Usa tuplas cuando:

- Necesitas agrupar tipos diferentes
- El número de elementos es pequeño (2-4)
- Los datos son temporales

Usa Vec cuando:

- El tamaño puede cambiar en tiempo de ejecución
- Trabajas con colecciones grandes
- Necesitas métodos como `push()`, `pop()`, `remove()`

Resumen rápido

```
// Creación
let arr = [1, 2, 3, 4, 5];
let repetido = [0; 10];

// Acceso
let primero = arr[0];
let seguro = arr.get(1); // Option<T>

// Desestructuración
let [a, b, ..] = arr;

// Mutabilidad
let mut m = [1, 2, 3];
m[0] = 10;

// Métodos
arr.len();
arr.first();
arr.last();
```

Rust


```
arr.contains(&3);

// Iteración
for x in arr { }
for (i, x) in arr.iter().enumerate() { }

// Multidimensional
let matriz = [[1, 2], [3, 4]];
```