Tipos de datos

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

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

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

Importancia del sistema de tipos en Rust

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

Tipos de Datos en Rust: Escalares y Compuestos

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.

- Compound Types

  Los tipos compuestos pueden agrupar múltiples valores en un solo tipo. Son estructuras que permiten organizar datos relacionados bajo una misma identidad.

Scalar Types

Tipos Enteros

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

Clasificación de los Enteros

Rust organiza los tipos enteros en dos categorías principales según su capacidad para representar números negativos:

Enteros con Signo (`i`)

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

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

Tabla 1: Rangos numéricos de los tipos enteros con signo

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

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

Enteros sin Signo (`u`)

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

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

Tabla 2: Rangos numéricos de los tipos enteros sin signo

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

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

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

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

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

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

Estos tipos se utilizan principalmente para:
- Indexar colecciones (vectores, arrays, slices)
- Representar tamaños de memoria y longitudes
- Realizar aritmética de punteros

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

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

Inferencia de Tipos Enteros

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

```rust
let numero = 42;        // Tipo inferido: i32
let negativo = -100;    // Tipo inferido: i32
let resultado = numero + negativo;  // i32
```

Especificación Explícita de Tipos

Anotación de Tipo

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

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

Sufijos de Tipo en Literales

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

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

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

Representación de Literales Enteros

Separadores Visuales

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

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

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

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

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

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

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

Modo debug:

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

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

Modo release:

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

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

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

```rust
let x: u8 = 255;

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

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

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

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

Conversiones entre Tipos Enteros

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

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

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

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

Constantes de Rango

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

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

Operaciones y métodos comunes

```rust
let x: i32 = 42;

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

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

Tabla 4: Operadores aritméticos

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

Tabla 5: Operadores de asignación compuesta

Tipos de Punto Flotante

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

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

Tipos Disponibles

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

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

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

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

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

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

Inferencia de Tipos

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

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

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

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

Limitaciones de Precisión

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

Ejemplo clásico de imprecisión:

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

Valores Especiales

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

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

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

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

Verificación de Valores Especiales

Rust proporciona métodos para detectar estos casos:

```rust
let valor = 1.0 / 0.0;

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

Constantes y Límites

Todos los tipos de punto flotante proporcionan constantes útiles:

```rust
// Límites numéricos
println!("f32 mínimo: {}", f32::MIN);           // -3.4028235e38
println!("f32 máximo: {}", f32::MAX);           // 3.4028235e38
println!("f64 mínimo: {}", f64::MIN);           // -1.7976931348623157e308
println!("f64 máximo: {}", f64::MAX);           // 1.7976931348623157e308
```

Operaciones Matemáticas

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

Redondeo

```rust
let valor = 3.7;

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

Operaciones con Signo

```rust
let negativo = -15.8;

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

Potencias y Raíces

```rust
let base = 4.0;

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

Descomposición de Valores

```rust
let numero:f32 = 42.75;

println!("Parte entera: {}", numero.trunc());           // 42.0
```

```rust
println!("Parte fraccionaria: {}", numero.fract());     // 0.75
```

Formato de Salida

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

```rust
let valor = 123.456789;

// Controlar decimales
println!("2 decimales: {:.2}", valor);          // 123.46
println!("5 decimales: {:.5}", valor);          // 123.45679

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

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

Conversiones entre Tipos

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

```rust
let x: f64 = 3.14159265359;

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

let a: f32 = 42.5;

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
```