

## 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;
```

#### 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;
```

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

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

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

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

Estos tipos se utilizan principalmente para:

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

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

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
```

## 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;
```

### 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
```

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;
```

### Bases Numéricicas

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)
```

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
```

### 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
```

#### 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)
```

### 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)
```

## 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)
```

**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);
```

## 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());
```

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