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 ella. 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.

Por ejemplo, una variable declarada como i32 solo puede almacenar números enteros de 32 bits, y no puede ser utilizada como una cadena de texto o un número decimal sin una conversión explícita.

```
let numero: i32 = 10;
```

Los tipos de datos son fundamentales en Rust por varias razones clave:

- Seguridad en tiempo de compilación

  Rust detecta muchos errores comunes como desbordamientos, usos incorrectos de memoria, tipos incompatibles, antes de que el programa se ejecute. Esto reduce significativamente errores en producción.

- Prevención de errores de memoria

  El sistema de tipos de Rust trabaja junto con el modelo de ownership, borrowing y lifetimes para garantizar seguridad de memoria sin necesidad de un recolector de basura.

- Rendimiento

  Al conocer los tipos en tiempo de compilación, el compilador puede generar código altamente optimizado, comparable o superior a C y C++.

- Claridad y mantenibilidad del código

  Los tipos hacen que el código sea más explícito, fácil de entender y mantener, especialmente en proyectos grandes.

Scalar Types

Integer Types

Los enteros son tipos de datos numéricos que representan valores sin parte decimal. En Rust, los enteros están diseñados para ser explícitos en tamaño y signo, lo que permite un control preciso sobre el uso de memoria y el comportamiento del programa.

Clasificación de los enteros en Rust

Rust divide los enteros en dos grandes categorías:

1. Enteros con signo “i”

   Los enteros con signo pueden representar valores positivos, negativos y el cero. Utilizan un bit para el signo y siguen la representación en complemento a dos.

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}	$2^{127} - 1$
isize	-2^{31} o -2^{63}	$2^{31} - 1$ o $2^{63} - 1$

Tabla 1: Rangos numéricos de los tipos enteros con signo.

Ejemplo:

```
let temperatura: i8 = -128;
temperatura = 127
```

2. Enteros sin signo “u”

   Los enteros sin signo solo pueden representar valores positivos o cero, ya que no reservan bits para el signo. Esto les permite cubrir un rango positivo mayor usando la misma cantidad de bits.

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	$2^{128} - 1$
usize	0	$2^{32} - 1$ o $2^{64} - 1$

Tabla 2: Rangos numéricos de los tipos enteros sin signo.

Ejemplo:

```
let mut temperatura: u8 = 0;
temperatura = 255;
```

Integer Overflow

Cuando un entero excede su rango permitido ocurre un desbordamiento.

```
let x: u8 = 255;
let y = x + 1; // overflow
```

- En modo debug, Rust detiene el programa con un error.
- En modo release, el valor hace wrap around volviendo a 0.

Inferencia de tipos enteros

Rust infiere tipos enteros cuando es posible:

```
let numero = 1000; // i32 por defecto
let numero = 1000i32;
let numero:i32 = 1_000;
```

El tipo por defecto es i32 porque ofrece un buen equilibrio entre rendimiento y rango.

Notación numérica

Rust permite diferentes bases numéricas:

```
let decimal = 255;       // base 10
let hexadecimal = 0xff;  // base 16
let octal = 0o377;       // base 8
let binario = 0b1111_1111; // base 2
let byte: u8 = b'A';     // byte literal (ASCII)
```

Conversión entre enteros

Rust no convierte tipos enteros automáticamente, debes hacerlo a mano:

```
let a: u16 = 1500;
let b: u32 = a as u32;  // conversión explícita
```

Rangos predefinidos

Todos los tipos enteros tienen constantes MIN y MAX:

```
println!("u8: {} - {}", u8::MIN, u8::MAX);
println!("i16: {} - {}", i16::MIN, i16::MAX);
```