VNCKey

RUST PERÚ

Diviértete con Rust

Viaja al futuro con el poder de Rust

- Programación de Sistemas
- Zero-Cost Abstractions
- Memory Safety

Alexander Villanueva

Contenido

Primeros pasos

¿Qué es Rust?

Rust es un lenguaje de sistemas moderno, diseñado para garantizar seguridad de memoria sin recolector de basura y rendimiento comparable a C y C++.

Su desarrollo comenzó en 2006 por Graydon Hoare y fue patrocinado por Mozilla, culminando con su primera versión estable en mayo de 2015.

Rust resuelve problemas clásicos de lenguajes de bajo nivel como punteros nulos, doble liberación y condiciones de carrera **data races**, mediante un sistema de propiedad **Ownership** y un verificador de préstamos **Borrowing** que el compilador valida en tiempo de compilación.

Rust no solo permite crear aplicaciones completas desde cero, puesto que su diseño de abstractions y memory safety lo convierte en una opción atractiva para reescribir componentes críticos en ecosistemas ya existentes.

Ejemplos concretos:

1. Firefox sustituyó partes de su motor CSS con Servo, reduciendo errores de memoria en un 70 %.
2. Discord migró su servicio de voz a Rust y dividió por 10 el uso de CPU frente a la versión en Go.
3. Dropbox reescribió su sistema de sincronización con Rust + Tokio, logrando menos latencia y menor consumo de memoria sin aumentar el coste de hardware.

Estas migraciones demuestran que Rust puede aumentar el rendimiento y reducir costes operativos sin sacrificar seguridad ni productividad.

La seguridad garantizada por su diseño ha generado una confianza excepcional entre los desarrolladores. De manera ininterrumpida desde 2016, Rust ha sido votado como el “lenguaje de programación más querido” en la encuesta anual de Stack Overflow.

Esta popularidad y la alta satisfacción de los desarrolladores son un fuerte indicador de su productividad y baja curva de frustración una vez dominado.

El compilador de Rust puede parecer exigente al principio, pero está diseñado para detectar errores *antes* de que tu código se ejecute. Esto significa menos bugs en producción y noches más tranquilas.

Además, Rust:

- **Compila directamente a código máquina**: Sin overhead de runtime ni máquinas virtuales
- **Soporta múltiples hilos sin data races**: El compilador garantiza que tu código concurrente es seguro
- **Elimina categorías enteras de bugs**: Muchos errores de concurrencia simplemente no pueden ocurrir

Por qué las empresas eligen Rust

Las razones principales que impulsan esta adopción son:

1. **Menos bugs en producción**: El compilador atrapa errores antes del despliegue
2. **Mayor rendimiento**: Comparable a C/C++ sin sacrificar seguridad
3. **Menor uso de recursos**: Traduce en ahorro de costos de infraestructura
4. **Código más mantenible**: Las garantías del lenguaje facilitan refactorings grandes

Instalación de Rust

Rust se instala mediante rustup, el instalador oficial que gestiona versiones del compilador, toolchains y herramientas asociadas. Este proceso es prácticamente idéntico en todos los sistemas operativos.

Linux y macOS

1. Abre una terminal.
2. Ejecuta el siguiente comando para descargar e iniciar el instalador de Rust:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```
Terminal

Este comando descarga el instalador de **rustup**, que configurará:

1. El compilador Rust **rustup**
2. **Cargo**, el gestor de paquetes y herramientas de construcción
3. La documentación estándar
4. Las herramientas necesarias para trabajar con Rust

Durante el proceso, se te preguntará si deseas continuar con la instalación por defecto o personalizarla.

La opción recomendada es presionar Enter para aceptar la configuración estándar.

Verás algo como esto:

```
--snip--
Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload your PATH environment variable to include
Cargo's bin directory ($HOME/.cargo/bin).

To configure your current shell, you need to source
the corresponding env file under $HOME/.cargo.
```
Terminal

Reinicia tu terminal o ejecuta:

```
source $HOME/.cargo/env
```
Terminal

Windows

Rust en Windows requiere el compilador **MSVC** de Microsoft para generar archivos ejecutables **.exe**. Debes instalar **Visual Studio Build Tools** antes de instalar Rust.

1. Instalar Visual Studio Build Tools

   Descarga Visual Studio desde:

   https://visualstudio.microsoft.com/es/downloads/

2. Componentes requeridos

   Rust necesita el workload **Desktop Development with C++** que incluye todas las herramientas esenciales para compilar programas nativos.

3. Instalación minimalista

Si no planeas desarrollar en C++ y quieres instalar solo lo indispensable:

Ve a la pestaña **Componentes individuales** y selecciona unicamente:

- **MSVC v143 – VS 2022 C++ x64/x86 build tools**
- **Windows 11 SDK (10.0.22621.0)**

Después de seleccionar los componentes, haz clic en Instalar y espera a que finalice.

4. Descargar Rust desde la página oficial

   Accede a la página oficial:

   https://www.rust-lang.org/tools/install

   Aquí encontrarás tres instaladores según la arquitectura de tu sistema:

   - rustup-init.exe (32 bits)
   - rustup-init.exe (x64 / 64 bits)
   - rustup-init.exe (ARM64)

   Selecciona el que corresponda a la arquitectura de tu sistema.

5. Ejecutar el instalador

   Ejecuta el archivo **rustup-init.exe** y sigue las instrucciones de la consola.

   Cuando aparezca el menú, simplemente presiona Enter para la instalación estándar:

   Terminal
   ```
   --snip--
   1) Proceed with standard installation (default - just press enter)
   2) Customize installation
   3) Cancel installation
   >1
   ```

   En la mayoría de casos es necesario cerrar y volver a abrir la terminal para que Windows reconozca los nuevos comandos **rustc cargo rustup** y sus todas sus herramientas.

6. Verificación de la Instalación

   Ejecuta los siguientes comandos en tu terminal:

   Terminal
   ```
   rustc --version  # rustc 1.89.0 (29483883e 2025-08-04)
   cargo --version  # cargo 1.89.0 (c24e10642 2025-06-23)
   rustup --version # rustup 1.27.0 (2024-08-14)
   ```

7. Solución de Problemas Comunes

   1. Linux/macOS: “comando no encontrado”

      Si después de instalar Rust sigues viendo command not found:

      Terminal
      ```
      # Verifica que $HOME/.cargo/bin esté en tu PATH
      echo $PATH | grep .cargo

      # Si no aparece, añádelo manualmente
      echo 'export PATH="$HOME/.cargo/bin:$PATH"' >> ~/.bashrc
      ```

```
source ~/.bashrc
```

)

1. Windows: “rustc no se reconoce como comando”
   - Reinicia tu terminal. Este es el problema más común.
   - Verifica que la instalación se completó sin errores.
   - Verifica manualmente que existe:

```
dir $env:USERPROFILE\.cargo\bin\rustc.exe
```
Terminal

Gestión de Versiones con Rustup

1. Actualizar Rust

```
rustup update
```
Terminal

1. Cambiar entre canales

```
rustup default stable   # Canal estable (recomendado)
rustup default beta     # Canal beta
rustup default nightly  # Canal nightly (experimental)
```
Terminal

No necesitas instalar **nightly** a menos que uses características experimentales o herramientas específicas que lo requieran.

Cargo

Cargo es el sistema de compilación y gestor de paquetes oficial de Rust. Es una herramienta fundamental que simplifica todo el ciclo de desarrollo, desde la creación de proyectos hasta la gestión de **dependencias** y la **compilación**.

Cargo se instala automáticamente cuando instalas Rust a través de rustup.

- Crea y estructura proyectos
- Compila tu código
- Descarga y gestiona dependencias
- Ejecuta tests
- Genera documentación
- Publica paquetes en crates.io

Compilación y Ejecución en Rust

La ejecución de código en Rust puede realizarse mediante dos vías fundamentales:
- Compilador directo, rustc, para tareas sencillas.
- Cargo la herramienta estándar de gestión de proyectos, indispensable para el desarrollo moderno.

Usando rustc

rustc es el compilador oficial de Rust. Es el programa que transforma tu código fuente (archivos .rs) en ejecutables que tu computadora puede ejecutar.

Entender **rustc** te ayuda a comprender mejor lo que sucede “bajo el capó”.

Imagina que escribes una receta en español, pero tu horno solo entiende instrucciones en lenguaje de máquina. El compilador **rustc** es el traductor que convierte tu receta **código Rust** en instrucciones que el horno **CPU** puede ejecutar.

Proceso de Compilación

Diagrama 1: Flujo de compilación en Rust: desde el código fuente hasta el ejecutable

Como vemos en el Diagrama 1, el compilador `rustc` es el encargado de transformar nuestro código.

Fases del Proceso

1. Paso 1: Creación del Módulo Fuente

   Todo comienza con el código fuente, que tradicionalmente lleva la extensión .rs.

   ```rust
   fn A main B() {
       println!("Compilador directo rustc."); //Archivo: main.rs
   }
   ```

   - A:

     Define la función principal y obligatoria de un programa ejecutable en Rust.

   - B:

     Es un nombre reservado y especial que el compilador de Rust y el sistema operativo buscan para saber dónde empezar a ejecutar el código.

2. Paso 2: Compilación

   Desde la terminal, se invoca a rustc, apuntando al archivo de entrada. El compilador lee el código y genera un archivo binario ejecutable en el mismo directorio.

   ```
   rustc main.rs
   ```

   En este proceso, rustc maneja internamente la verificación de tipos, el borrow checker y la generación del código máquina optimizado, utilizando LLVM.

3. Ejecución

   Esto genera un ejecutable.

   - Windows:

     ```
     .\main.exe
     ```

   - Linux/macOS:

     ```
     ./main
     ```

   Resultado:

   ```
   Compilador directo rustc.
   ```

Tu Primer Proyecto con Cargo

Crear un nuevo proyecto

```
# Crear un proyecto binario (aplicación)
```

```
cargo new a hola_mundo b
```

- a: Crear un nuevo proyecto Rust
- b: Nombre del proyecto

```
# Entrar al directorio                                    Terminal
cd hola_mundo
```

Estructura creada:

```
hello_world/ Raíz del proyecto                                Output
├── Cargo.lock Registra las versiones específicas
├── Cargo.toml Define las dependencias
├── src/
│   └── main.rs Código fuente principal
└── target/ Destino de la Compilación
    └── debug/
        ├── build/
        └── deps/
            └── hello_world El ejecutable de tu aplicación
```

Anatomía del Proyecto: Cargo.toml

Contenido inicial de Cargo.toml

```toml
[package]                                                     toml
name = "hola_mundo" Nombre del proyecto
version = "0.1.0" Versión siguiendo                    } (1) Metadatos
edition = "2021" Edición estable de Rust 2021

[dependencies] crates
# Ejmplos
# rand = "0.8.5" Permite generar números aleatorios    } (2) Librerías externas

# serde = "1.0.130" Permite serializar y deserializar datos
```

Punto de entrada: main.rs

```
fn main() A { C                                               Rust
    println! B("Hola, Rust!");
} C
```

1. A: Define la función principal del programa
2. B: Es una macro que imprime texto en la consola

3. C: Delimitan el bloque de código de la función

Compilar y Ejecutar

```
cargo run
```
Terminal

```
Compiling hola_mundo v0.1.0 (/ruta/hola_mundo) A
 Finished `dev` profile [unoptimized + debuginfo] target(s) in 3.42s B
  Running `target/debug/hola_mundo` C
Hola, Rust! D
```
Terminal

1. A: Cargo compila el proyecto (solo la primera vez o si hay cambios)
2. B: Perfil de compilación: dev (desarrollo, sin optimizaciones)
3. C: Ruta del ejecutable que se está ejecutando
4. D: Output de tu programa

Si no modificaste el código, la segunda ejecución será instantánea:

```
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
 Running `target/debug/hola_mundo`
Hola, Rust!
```
Terminal

Variables con Rust

Let

```rust
fn main() {
  let A edad B = 25 C;
  println!("Mi edad es {}", edad);
}
```
Rust

1. A: La palabra reservada **let** se usa para declarar variables inmutables por defecto
2. B: Nombre de la variable **edad**
3. C: Asignación de tipo de valor entero **25**

```
Mi edad es 25
```
Output

Que pasa si queremos modificar el valor de una variable inmutable?

```rust
fn main() {
  let edad = 25;
  edad = 26; ERROR!!
  println!("Mi edad es {}", edad);
}
```
Rust

Gracias a herramientas inteligentes como rust-analyzer y rustc, nuestros editores de código pueden analizar información avanzada e interactiva a través del Language Server Protocol (LSP). De esta manera, es posible visualizar errores, comprender por qué el código es incorrecto e incluso recibir sugerencias automáticas para corregirlo.

```
error[E0384]: No se puede asignar dos veces a la variable inmutable
`edad`
 --> src/main.rs:3:3
3 |   edad = 26;
  |
help: consider making this binding mutable
  |
2 |   let mut edad = 25;
  |       +++
For more information about this error, try `rustc --explain E0384`.
```
Terminal

1. El Código de Error: [E0384]

   Este es el identificador único y universal del problema.

   Con documentacion **https://doc.rust-lang.org/error_codes/error-index.html** exacta para el tipo de error y explicacion detallada de ese problema.

variables mutables

Si necesitas cambiar el valor de una variable, debes declararla explícitamente como mutable usando **let mut**.

```rust
fn main() {
  let mut A carro = "Toyota";
  println!("Mi carro es {}", carro);
  carro = "Honda";
  println!("Mi nuevo carro es {}", carro);
}
```
Rust

1. A: La palabra reservada **let mut** se usa para declarar variables mutables que cambian su valor a lo largo del programa.

Shadowing

El shadowing permite declarar una nueva variable con el mismo nombre que una anterior.

La nueva variable “sombrea” a la anterior.

```rust
fn main() {
  let mut edad = 25;
  println!("Mi edad es {}", edad);
  let edad = "Mi edad es 35"; A
  println!("{}", edad);
}
```
Rust

- A:
  - Rust permite declarar una nueva variable con el mismo nombre que una anterior.
  - La nueva variable “sombrea” a la anterior.
  - Shadowing permite cambiar el tipo de una variable.

```
Mi edad es 25
Mi edad es 35
```
Output

Lo que no se puede hacer es cambiar el tipo de una variable sin "sombrear".

```rust
let mut texto = "Hola";
texto = 5; ERROR!
```
Rust

Scopes

El scope determina dónde una variable es válida en tu código. En Rust, el scope está definido por llaves { }.

```rust
fn main() {
  let x = 5;
  println!("Valor de x: {x}");
  {
    let x = x * 2;
    let y = x;
    println!("Dentro del scope x: {x}");
    println!("Dentro del scope y: {y}");
  } A

  //println!("Valor de y: {y}"); ERROR!
  println!("Valor de x: {x}");
}
```
Rust

- A:
  - La variable x y y son válidas dentro del scope en el que fueron declaradas.
  - Terminado el scope, las variables x y y son liberadas.
  - Ya no se pueden usar fuera del scope en el que fueron declaradas.

Nota: Puedes crear scopes anidados.

```
Valor de x: 5
Dentro del scope x: 10
Dentro del scope y: 10
Valor de x: 5
```
Output

Constantes

Las constantes son valores globales que nunca cambian y deben tener un tipo explícito. No tienen una dirección de memoria fija. Se utiliza para valores que son absolutamente fijos y conocidos de antemano, como constantes matemáticas, límites, o configuraciones fijas.

```rust
const PI A : f64 B = 3.14159265359;

fn main() {
  const PI: f64 = 5.14; C
```
Rust

```
  println!("Valor de PI: {}", PI);
  //PI = 3.12; D
}
```

1. A:

   Las constantes siempre usan SCREAMING_SNAKE_CASE.

2. B:

   El tipo debe ser explícito :f64 en este caso flotante.

3. C:

   Rust permiten sombrear constantes con el mismo nombre.

4. D:

   Rust no permite mutar constantes.

```
Valor de PI: 5.14
```
Output

Valores estaticos

Las variables estáticas tienen una ubicación fija en memoria y viven durante toda la ejecución del programa. Se inicializan al inicio de la ejecución del programa (cuando el programa se carga, antes de que se ejecute la función main).

```rust
static PROTOCOLO_VERSION A: u8 = 2;

fn main() {
  // let PROTOCOLO_VERSION:u8 = 3; B
  // PROTOCOLO_VERSION: u8 = 8; C
  println!("Protocolo v{}", PROTOCOLO_VERSION);
}
```
Rust

1. A:

   Declaramos un valor estatico con **static**

2. B

   No podemos sombrear un valor estatico con el mismo nombre.

3. C

   No podemos mutar un valor estatico.

```
Protocolo v2
```
Output

Statements & Expressions

Una sentencia es una instrucción que realiza una acción y no devuelve un valor. En Rust, la mayoría de las sentencias terminan con un punto y coma ;.

Una expresión es cualquier pieza de código que se evalúa y devuelve un valor.

```rust
fn main() {
    let y = { A
        let z = 3; B
        z + 1 D
    }; A

    println!("y = {}", y); A
}
```

1. A:
   - Tenemos la primera Sentencia (statement) `let y = { ... };`, una unidad de ejecución que no produce un valor que pueda ser utilizado por otra parte del código.
   - Expresión Asignada: `{ ... }` La expresión de bloque se evalúa y devuelve un valor
2. B:
   - Tenemos la segunda Sentencia (statement) La sentencia **let** realiza la acción de vincular un valor a un nombre y nunca devuelve un valor, Rust evita side-effect oculto.

     `let y = (let x = 5); ERROR!!!!`

     Ejemplos:

     Javascript

     ```javascript
     let x = 1;
     let y = (x = 2, x++);   // y = 2; x = 3
     console.log(y, x);      // 2 3 = side-effect dentro de la expresión

     if (count = 0) { }   // 0 es falsy = nunca entra, pero *asigna*
     ```

     Python

     ```python
     b = 5
     a = (b := 1) + (b := 2)   # a = 2
     print("Valor de a es: ",a) #Valor de a es: 3
     print("Valor de b es: ",b) #Valor de b es: 2
     ```

   - Rust prohíbe que **let** devuelva valor y así evita bugs clásicos como `if (x = 5)`.
   - `println!` devuelve **unit type ()**, y la llamada como declaración.
   - Rust prohíbe que **let** sea una expresión para eliminar una clase entera de errores que sí existen en lenguajes donde la asignación devuelve valor.
   - La regla de oro en Rust es que la mayoría de las sentencias terminan con un `;` .
3. C:
   - Tenemos la primera Expresión (Expression) Cuando omites el punto y coma en la última línea de un bloque, le estás diciendo al compilador:

     "Quiero que el valor resultante de esta operación sea el valor de retorno de todo el bloque."

4. D:

   Por ultimo, tenemos un Statement

   Aunque la llamada a la macro println! es técnicamente una expresión (ya que se evalúa), su valor de retorno es el tipo unitario () (pronunciado “unit”).

```
y = 4
```
Output

En caso de poner ; al final se convierte en una sentencia (statement) y devuleve un unit type ().

```rust
fn main() {
  let y = {
    let z = 3;
    z + 1; A
  };
  println!("y = {:?}", y);
}
```
Rust

1. A:

   Devuelve unit type ()

```
y = ()
```
Output

Sistema de Memoria

La seguridad de memoria de Rust garantiza que un programa nunca acceda a memoria inválida ni cometa errores peligrosos al manejar recursos. En lenguajes como C y C++, esta responsabilidad recae por completo en el programador, lo que abre la puerta a introducir accidentalmente vulnerabilidades como desbordamientos de búfer, use-after-free, double free o data races.

Rust elimina estos problemas desde su diseño.

Todo esto sin la necesidad de un garbage collector y sin costo adicional en tiempo de ejecución. Este enfoque permite escribir software seguro y eficiente.

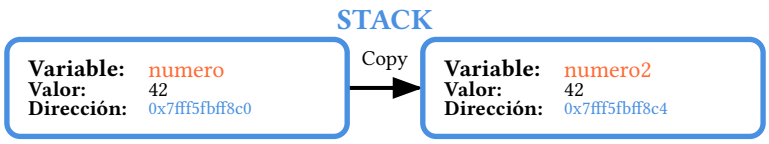
Los primeros conceptos pilares es comprender como se administra la memoria. Rust divide la memoria en dos grandes regiones Stack y Heap, su diferencia es esencial para evitar errores como:

- use-after-free
- double free
- dangling pointers
- data races

Stack

El Stack es una región de la memoria RAM para almacenar datos cuya vida útil y tamaño son conocidos en tiempo de compilación.

```rust
fn main() {
    let numero: u8 = 5;
    let numero2 = numero;
}
```
Rust



Los valores se copian porque `i32` implementa `Copy`

Diagrama 2: Representación del stack de memoria en Rust

Heap

El Heap es una región de la memoria RAM para almacenar datos cuya vida útil y tamaño son conocidos en tiempo de ejecución.

```rust
fn main() {
  let texto: String A = String::from("Rûst"); B
  let texto2: String = texto; C

  println!("{texto2}");
  //println!("{texto1}"); D
}
```

1. A:
   - String es un tipo de dato que se almacena en el Heap y su tamaño es desconocido en tiempo de compilación.
   - Se usa para representar texto dinámico, cuya longitud o contenido puede cambiar en tiempo de ejecución.
2. B:
   - Sirve para construir una String a partir de un &str por ejemplo un “Hola” u otros tipos convertibles, como otro mismo String.
   - Es equivalente a .to_string() en muchos casos, pero se prefiere String::from() por ser más explícito y genérico.
3. C:
   - texto tiene un comportamiento llamado Move cuando texto2 la consume.
   - Después de la asignación, texto ya no es válido, puesto que el compilador no te dejará usarlo, move semantics.
   - Primera regla de Ownership solo puede haber un único propietario y es por esta razon que Rust no permite que dos variables apunten al mismo dato en el Heap.
   - texto2 es ahora el propietario del String “Rûst”
   - println!(“{texto}”);
4. D:
   - El uso de la variable texto es invalido, ya que fue movida a texto2.

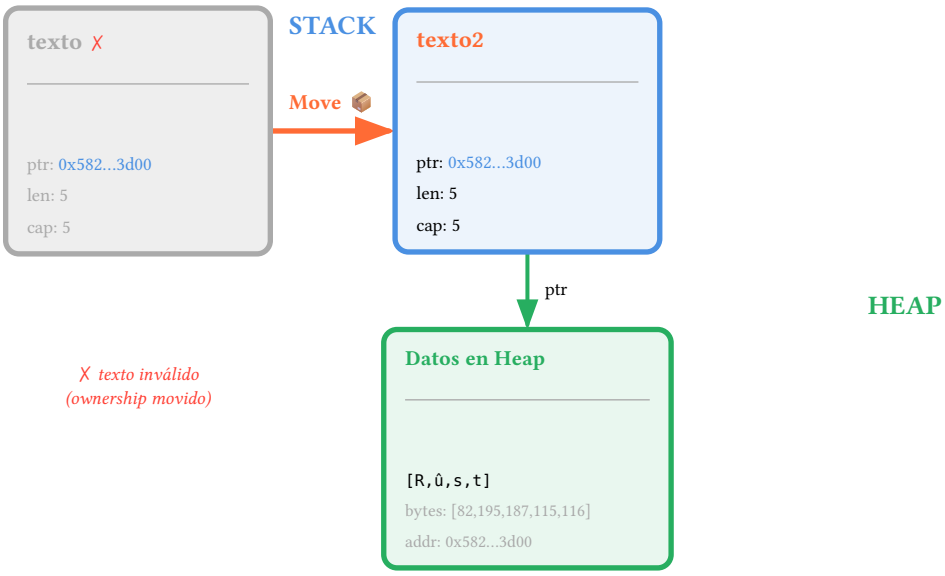


Diagrama 3: Move semántico: el ownership se transfiere de `texto` a `texto2`

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);
```