main.rs

```rust
fn main() {
    println!(«¡Bienvenido a Rust!»);
    let aprendizaje = vec![      «Ownership»,
        «Borrowing»,
        «Lifetimes»,
        «Traits», ]

    for tema in aprendizaje {
        println!(«📚 {}», tema);
    }
}
```

Rust Para Todos

Guía de Estudio Personal

M.H. Alberto

2025-12-23

Índice

1

Introducción a Rust

Rust es un lenguaje de programación de sistemas moderno que revoluciona la forma en que escribimos software de bajo nivel. Se enfoca en tres pilares fundamentales: seguridad, velocidad y concurrencia, ofreciendo un equilibrio único que ningún otro lenguaje había logrado antes.

1.1 ¿Qué es Rust?

Rust es un lenguaje de programación compilado, de tipado estático y multiparadigma. Fue diseñado para ser un lenguaje que prioriza la seguridad sin sacrificar el rendimiento. A diferencia de lenguajes como C y C++, Rust garantiza la seguridad de memoria en tiempo de compilación, eliminando toda una categoría de errores comunes que han plagado el desarrollo de software durante décadas.

El lenguaje combina ideas de programación funcional, orientada a objetos e imperativa, permitiendo a los desarrolladores escribir código expresivo y eficiente. Su sistema de tipos es rico y flexible, mientras que su compilador actúa como un asistente inteligente que detecta errores potenciales antes de que el código se ejecute.

Rust se utiliza en una amplia variedad de dominios: desde sistemas operativos y navegadores web hasta herramientas de línea de comandos, servicios web, aplicaciones embebidas y sistemas blockchain. Empresas como Mozilla, Microsoft, Google, Amazon, Facebook y Dropbox han adoptado Rust en proyectos críticos de producción.

1.2 ¿Por qué Rust?

Rust elimina clases enteras de errores en tiempo de compilación gracias a su innovador sistema de ownership y borrowing. Esto significa que:

- **No hay errores de segmentación**: El compilador verifica que todas las referencias a memoria sean válidas, eliminando los temidos «segmentation faults» que atormentan a los programadores de C y C++.
- **No hay condiciones de carrera en datos**: El sistema de tipos de Rust garantiza que no puedas tener referencias mutables e inmutables al mismo dato simultáneamente, previniendo race conditions en tiempo de compilación.
- **No hay punteros nulos sin manejar**: Rust no tiene valores nulos. En su lugar, utiliza el tipo `Option<T>` que obliga al programador a manejar explícitamente los casos donde un valor puede no existir.
- **No hay uso de memoria después de liberarla**: El compilador rastrea la vida útil de cada valor y garantiza que ninguna referencia sobreviva al dato al que apunta.

- **No hay desbordamientos de buffer**: Los accesos a arrays y slices se verifican en tiempo de ejecución en modo debug, y el código que accede directamente a memoria debe estar marcado explícitamente como `unsafe`.

1.2.1 Rendimiento comparable a C y C++

Rust ofrece abstracciones de costo cero, lo que significa que las características de alto nivel del lenguaje se compilan a código tan eficiente como si lo hubieras escrito manualmente en ensamblador. El compilador de Rust, basado en LLVM, realiza optimizaciones agresivas que producen binarios extremadamente rápidos.

A diferencia de lenguajes como Java, Python o JavaScript, Rust no tiene recolector de basura. Esto significa que no hay pausas impredecibles en tiempo de ejecución, lo que lo hace ideal para sistemas de tiempo real y aplicaciones de baja latencia.

1.2.2 Concurrencia sin miedo

Escribir código concurrente en Rust es significativamente más seguro que en otros lenguajes. El sistema de tipos previene condiciones de carrera en tiempo de compilación, permitiéndote usar múltiples hilos sin el temor constante de introducir bugs sutiles y difíciles de depurar.

Rust soporta múltiples modelos de concurrencia: desde hilos tradicionales del sistema operativo hasta async/await para operaciones de I/O asíncronas. El compilador te ayuda a escribir código paralelo correcto desde el primer intento.

1.2.3 Herramientas excepcionales

Rust viene con un ecosistema de herramientas moderno y cohesivo:

- **Cargo**: Un gestor de paquetes y sistema de compilación integrado que simplifica la gestión de dependencias, compilación, testing y publicación de bibliotecas.
- **rustfmt**: Un formateador automático de código que asegura un estilo consistente en todos los proyectos Rust.
- **Clippy**: Un linter avanzado que ofrece sugerencias para mejorar tu código y hacerlo más idiomático.
- **rust-analyzer**: Un servidor de lenguaje LSP que proporciona autocompletado inteligente, navegación de código y refactorizaciones en tu editor favorito.

1.2.4 Documentación de primera clase

La comunidad de Rust considera la documentación como parte integral del código. El compilador puede generar automáticamente documentación HTML a partir de comentarios especiales, y prácticamente todas las bibliotecas públicas están bien documentadas con ejemplos funcionales.

1.2.5 Comunidad acogedora

Rust es conocido por tener una de las comunidades más amigables y colaborativas en el mundo del desarrollo de software. El proyecto mantiene un código de conducta estricto y prioriza la inclusión y el respeto mutuo.

1.3 Historia de Rust

Rust comenzó como un proyecto personal de Graydon Hoare, un desarrollador canadiense que trabajaba en Mozilla Research. En 2006, frustrado por los bugs de seguridad de memoria que

encontraba repetidamente en software crítico, Hoare comenzó a diseñar un nuevo lenguaje que pudiera prevenir estos problemas en tiempo de compilación.

El nombre *Rust* (óxido en inglés) hace referencia a los hongos del orden Pucciniales, conocidos por ser extremadamente resistentes y robustos, metáfora apropiada para las ambiciones del lenguaje.

En 2009, Mozilla se interesó en el proyecto y comenzó a patrocinarlo oficialmente. La motivación principal era construir Servo, un motor de navegador experimental escrito completamente en Rust que pudiera aprovechar el paralelismo moderno de manera segura.

Durante este período, el lenguaje evolucionó dramáticamente. Se implementaron características fundamentales como el sistema de ownership, lifetimes, traits y el sistema de macros. El compilador inicial escrito en OCaml fue reemplazado por uno bootstrapped escrito en el propio Rust.

El 15 de mayo de 2015, se lanzó Rust 1.0, marcando la primera versión estable del lenguaje. Este fue un hito crucial que garantizaba estabilidad hacia atrás: el código que compilaba con Rust 1.0 seguiría compilando con versiones futuras del compilador.

Junto con 1.0, se estableció un modelo de lanzamiento predecible con nuevas versiones cada seis semanas, similar al proceso de Firefox. Este cadencia rápida permite incorporar mejoras continuamente sin romper el código existente.

Los años siguientes vieron una adopción masiva de Rust en la industria. Se introdujeron características importantes como:

- **2017**: Mejoras en el tiempo de compilación y ergonomía del lenguaje
- **2018**: Rust 2018 Edition con mejoras en módulos y async/await preliminar
- **2019**: Estabilización de async/await, transformando el desarrollo asíncrono en Rust
- **2020**: Rust 2020 Edition y la fundación del proyecto Rust Foundation

Durante este período, grandes empresas comenzaron a invertir seriamente en Rust. Microsoft adoptó Rust para componentes críticos de seguridad en Windows. Amazon Web Services creó equipos dedicados a Rust. Google comenzó a permitir Rust en el kernel de Android.

En febrero de 2021, se estableció la Rust Foundation como una organización independiente sin fines de lucro para administrar el proyecto. Los miembros fundadores incluyeron Mozilla, Amazon Web Services, Huawei, Google y Microsoft, cada uno contribuyendo financiamiento y recursos.

La fundación asegura que Rust tenga un futuro estable e independiente de cualquier compañía individual. Hoy, Rust continúa creciendo con miles de colaboradores y millones de usuarios en todo el mundo.

Rust ha sido votado como el «lenguaje más amado» en la encuesta anual de Stack Overflow durante ocho años consecutivos (2016-2023), un testimonio de la satisfacción de los desarrolladores que lo usan. Aquí tienes el **código completo en Typst**, listo para copiar y usar:

1.4 Crear un proyecto en Rust

En Rust contamos con un sistema llamado `cargo`, el cual nos permite crear y gestionar proyectos, además de compilar y ejecutar código Rust de forma sencilla. Desde la consola, escribimos el siguiente comando para crear un nuevo proyecto:

```
cargo new nombre_del_proyecto
```

Este comando generará una carpeta con la estructura básica del proyecto. Dentro del directorio `/src` encontraremos el archivo `main.rs`, que es donde se escribe el código principal del programa. La extensión `.rs` es la utilizada por Rust para sus archivos fuente. Para ejecutar el proyecto, utilizamos nuevamente `cargo` con el siguiente comando:

```
cargo run
```

Esto compilará el código y ejecutará el programa automáticamente.

1.5 Hola mundo

En prácticamente todos los lenguajes de programación existe un programa clásico que, al ejecutarse, imprime en pantalla el mensaje `¡Hola mundo!`. En Rust no es la excepción, y también comenzaremos con este ejemplo.

Para ello, es importante saber que Rust utiliza **macros**. Una macro es una construcción que genera código en tiempo de compilación, es decir, antes de que el programa se ejecute.

En Rust, `println!()` —la forma que usamos para imprimir mensajes en la consola— no es una función, como ocurre en muchos otros lenguajes. En lugar de trabajar con valores, trabaja con fragmentos de código, también conocidos como **tokens**. Por esta razón, `println!()` es una macro declarativa, lo cual se identifica fácilmente porque termina con el símbolo `!`.

Para crear nuestro primer programa, utilizaremos el archivo `hola_mundo.rs` y escribiremos el siguiente código:

```
fn main() {
    println!("¡Hola mundo!");
}
```

Al ejecutar este archivo, Rust imprimirá el mensaje `¡Hola mundo!` en la consola, confirmando que nuestro entorno funciona correctamente.

1.6 Variables inmutables y mutables

En Rust, para definir una variable utilizamos la palabra reservada `let`. Por defecto, las variables en Rust son **inmutables**, lo que significa que, una vez asignado un valor, no puede modificarse posteriormente. Si intentamos reasignar un valor distinto, el compilador nos mostrará un error.

Comencemos viendo cómo definir una variable inmutable. Para ello, crearemos un nuevo archivo llamado `variable_inmutable.rs`:

```rust
fn main() {
    let saludo = "Hola mundo";
    println!("La variable saludo es {}", saludo);
}
```

En este ejemplo utilizamos la macro `println!`, la cual nos permite imprimir mensajes en la consola. Además de mostrar texto, también puede imprimir el valor de una variable. Para ello, Rust utiliza un sistema de interpolación mediante llaves `{}`, donde el valor de la variable se pasa como argumento después de una coma.

En este caso, la variable `saludo` imprime en consola el valor `"Hola mundo"`.

Si intentamos reasignar un valor a la variable `saludo`, Rust generará un error de compilación, ya que las variables son inmutables por defecto:

```rust
fn main() {
    let saludo = "Hola mundo";
    println!("La variable saludo es {}", saludo);
    saludo = "Mi nombre es";
}
```

Para permitir que una variable pueda cambiar su valor, debemos indicar explícitamente que es **mutable** usando la palabra reservada `mut` después de `let`.

El ejemplo anterior se puede corregir de la siguiente manera, para ello escribiremos en un archivo llamado `variable_mutable.rs` la siguiente sentencias:

```rust
fn main() {
    let mut saludo = "Hola mundo";
    println!("La variable saludo es {}", saludo);
    saludo = "Saludando al mundo";
    println!("La variable saludo reasignada es {}", saludo);
}
```

De esta forma, Rust nos permite modificar el valor de la variable sin generar errores, manteniendo un control explícito sobre la mutabilidad del código.

2

Tipos de datos

Rust es un lenguaje fuertemente tipado, lo que significa que cada variable tiene un tipo bien definido y conocido en tiempo de compilación. Sin embargo, esto no implica que el programador tenga que escribir explícitamente el tipo en cada declaración.

El compilador de Rust cuenta con un potente sistema de inferencia de tipos, capaz de deducir el tipo correcto a partir del valor asignado y del contexto en el que se utiliza la variable. Gracias a esto, el código puede mantenerse limpio y conciso sin perder seguridad ni precisión en el tipado.

2.1 Inferencia de tipos

Rust puede inferir automáticamente los tipos más comunes:

```rust
fn main() {
    let x = 5;          // Rust infiere: i32
    let y = 3.14;       // Rust infiere: f64
    let z = true;       // Rust infiere: bool
    let a = 'A';        // Rust infiere: char (UNICODE)
    let b = "Hola";     // Rust infiere: &str (string slice)
}
```

2.2 Tipos enteros con signo

Los enteros con signo pueden representar números positivos y negativos. Su identificador comienza con `i`:

Tipo	Bits	Rango
`i8`	8	-128 a 127
`i16`	16	$-32,768$ a $32,767$
`i32`	32	$-2,147,483,648$ a $2,147,483,647$
`i64`	64	-2^{63} a $2^{63}-1$
`i128`	128	-2^{127} a $2^{127}-1$
`isize`	arquitectura	Depende del sistema (32 o 64 bits)

```rust
fn main() {
    let a: i8 = -128;
    let b: i16 = -32_000;
    let c: i32 = -10;
    let d: i64 = 1_000_000;
    let e: i128 = -123_456_789_012_345;
    let f: isize = -100; // Tamaño según arquitectura
}
```

2.3 Tipos enteros sin signo

Los enteros sin signo solo representan números positivos. Su identificador comienza con `u`:

Tipo	Bits	Rango
`u8`	8	0 a 255
`u16`	16	0 a 65,535
`u32`	32	0 a 4,294,967,295
`u64`	64	0 a $2^{64} - 1$
`u128`	128	0 a $2^{128} - 1$
`usize`	arquitectura	Depende del sistema (32 o 64 bits)

```rust
fn main() {
    let a: u8 = 255;
    let b: u16 = 65_000;
    let c: u32 = 1_000_000;
    let d: u64 = 18_446_744_073_709_551_615;
    let e: u128 = 340_282_366_920_938_463_463_374_607_431_768_211_455;
    let f: usize = 100; // Usado para índices de arrays
}
```

2.4 Tipos de punto flotante

Rust proporciona dos tipos para números decimales, siguiendo el estándar IEEE-754:

Tipo	Bits	Precisión
`f32`	32	Precisión simple (7 dígitos decimales)
`f64`	64	Precisión doble (15 dígitos decimales)

```rust
fn main() {
    let pi: f32 = 3.14159;           // 32 bits
    let euler: f64 = 2.718281828459; // 64 bits (por defecto)

    // Operaciones científicas
    let grande = 1.5e10;  // 15,000,000,000
    let pequeño = 3.2e-5; // 0.000032
}
```

2.5 Tipo booleano

El tipo `bool` representa valores de verdad y solo puede ser `true` o `false`:

```rust
fn main() {
    let es_mayor: bool = true;
    let es_menor: bool = false;

    // Resultado de comparaciones
    let resultado = 5 > 3;  // true
    let otro = 10 == 5;     // false
}
```

2.6 Tipo carácter

El tipo `char` representa un carácter Unicode individual. Se define con comillas simples y ocupa 4 bytes:

```rust
fn main() {
    let letra: char = 'A';
    let emoji: char = '😀';
    let kanji: char = '字';
    let simbolo: char = '∞';

    // Caracteres especiales
    let tabulacion: char = '\t';
    let nueva_linea: char = '\n';
    let unicode: char = '\u{1F980}'; // 🦀 (cangrejo)
}
```

2.7 Tipos de texto

2.7.1 String slice (`&str`)

Un `&str` es una referencia inmutable a una secuencia de texto UTF-8. Es el tipo más básico para texto:

```
fn main() {
    let saludo: &str = "Hola, mundo";
    let multilinea: &str = "Primera línea
Segunda línea
Tercera línea";

    // Literales crudos (raw strings)
    let ruta: &str = r"C:\Users\nombre\archivo.txt";
}
```

2.7.2 String (`String`)

`String` es un tipo de texto dinámico, mutable y de tamaño variable almacenado en el heap:

```
fn main() {
    let mut texto: String = String::from("Hola");
    texto.push_str(", mundo"); // "Hola, mundo"
    texto.push('!');           // "Hola, mundo!"

    // Creación de Strings
    let s1 = String::new();
    let s2 = "contenido".to_string();
    let s3 = String::from("más texto");
}
```

2.8 Tuplas

Las tuplas agrupan valores de diferentes tipos en un solo tipo compuesto:

```
fn main() {
    let persona: (&str, i32, f64) = ("Alice", 30, 1.65);

    // Acceso por índice
    let nombre = persona.0;  // "Alice"
    let edad = persona.1;    // 30
    let altura = persona.2;  // 1.65
```

```
    // Desestructuración
    let (n, e, a) = persona;

    // Tupla vacía (unit)
    let vacio: () = ();
}
```

2.9 Arrays

Los arrays tienen tamaño fijo y todos sus elementos deben ser del mismo tipo:

```
fn main() {
    // Declaración explícita: [tipo; tamaño]
    let numeros: [i32; 5] = [1, 2, 3, 4, 5];

    // Inicialización con valor repetido
    let ceros: [i32; 10] = [0; 10]; // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

    // Acceso por índice
    let primero = numeros[0];  // 1
    let ultimo = numeros[4];   // 5

    // Arrays multidimensionales
    let matriz: [[i32; 3]; 2] = [
        [1, 2, 3],
        [4, 5, 6],
    ];
}
```

2.10 Slices

Los slices son referencias a secuencias contiguas de elementos en una colección:

```
fn main() {
    let numeros: [i32; 5] = [1, 2, 3, 4, 5];

    // Slice de array
    let todos: &[i32] = &numeros[..];      // [1, 2, 3, 4, 5]
    let algunos: &[i32] = &numeros[1..4];  // [2, 3, 4]
    let primeros: &[i32] = &numeros[..3];  // [1, 2, 3]
    let ultimos: &[i32] = &numeros[2..];   // [3, 4, 5]

    // Slice de String
```

```
    let texto = String::from("Hola mundo");
    let palabra: &str = &texto[0..4];  // "Hola"
}
```

2.11 Anotación explícita de tipos

Aunque Rust infiere tipos, puedes especificarlos explícitamente:

```
fn main() {
    // Sin anotación (inferido)
    let x = 5;
    let y = 3.14;

    // Con anotación explícita
    let a: i32 = 5;
    let b: f64 = 3.14;
    let c: bool = true;
    let d: char = 'R';
    let e: &str = "texto";

    // Útil en ambigüedades
    let numero = "42".parse::<i32>().unwrap();
}
```

2.12 Conversión entre tipos

Rust requiere conversiones explícitas entre tipos numéricos:

```
fn main() {
    let entero: i32 = 10;
    let flotante: f64 = entero as f64;  // 10.0

    let grande: i64 = 1000;
    let pequeño: i32 = grande as i32;   // 1000

    // Conversión de caracteres
    let letra: char = 'A';
    let codigo: u32 = letra as u32;     // 65

    // Parsing de strings
    let texto = "123";
    let numero: i32 = texto.parse().unwrap();  // 123
}
```

2.13 Separadores numéricos

Rust permite usar guiones bajos como separadores para mejorar la legibilidad:

```rust
fn main() {
    let millon = 1_000_000;
    let binario = 0b1111_0000;
    let hexadecimal = 0xFF_FF_FF;
    let flotante = 123_456.789_012;

    println!("Un millón: {}", millon);
}
```