Diviértete con Rust 🦀
Viaja al futuro con el poder de Rust

Primera edición: Diciembre 2024

Publicado por VNCKey
En colaboración con la comunidad Rust Perú 🇵🇪
Composición tipográfica con Typst
Fuente: Crimson Pro

Diviértete con Rust
Viaja al futuro con el poder de Rust

Primera edición: Diciembre 2024

Publicado por VNCKey
Composición tipográfica con Typst
Fuente: Crimson Pro

Para todos aquellos que quieren dominar
la programación de sistemas del futuro.

Prefacio

Este libro nace de la pasión por Rust y la necesidad de hacer su aprendizaje más accesible y divertido. A lo largo de estas páginas, descubrirás no solo un lenguaje de programación, sino una nueva forma de pensar sobre la seguridad, el rendimiento y la expresividad del código.

A quién va dirigido este libro

Este libro es para ti si:

- Quieres aprender programación de sistemas
- Vienes de otros lenguajes como Python, JavaScript o C++
- Te interesa la seguridad y el rendimiento
- Buscas un lenguaje moderno y expresivo

Cómo usar este libro

Cada capítulo está diseñado para construir sobre los anteriores. Te recomiendo seguir el orden, pero si ya tienes experiencia, puedes saltar a los temas que más te interesen.

Introducción a Rust

Rust es un lenguaje de programación de sistemas que se enfoca en tres pilares fundamentales: seguridad, velocidad y concurrencia. Desarrollado inicialmente por Mozilla, hoy es uno de los lenguajes más amados por la comunidad de desarrolladores.

¿Por qué Rust?

En el panorama actual de la programación, Rust destaca por varias razones fundamentales. A diferencia de lenguajes como C o C++, Rust garantiza seguridad de memoria sin necesidad de un recolector de basura.

Los principales beneficios de Rust incluyen:

1. **Seguridad de memoria**: El sistema de ownership previene errores comunes como use-after-free y data races.
2. **Rendimiento comparable a C/C++**: Zero-cost abstractions significa que no pagas por lo que no usas.
3. **Concurrencia sin miedo**: El compilador garantiza que tu código concurrente es seguro.
4. **Excelente tooling**: Cargo, rustfmt, clippy y más herramientas de primera clase.

   Rust tiene una curva de aprendizaje pronunciada al inicio. El concepto de ownership puede ser confuso, pero una vez lo dominas, todo cobra sentido.

Tu primer programa en Rust

Empecemos con el tradicional "Hello, World!". Crea un nuevo proyecto con:

```
cargo new hello_rust
cd hello_rust
```

Abre `src/main.rs` y verás:

```
fn main() {
    println!("Hello, world!");
}
```

Este simple programa introduce varios conceptos:

- `fn` declara una función
- `main` es el punto de entrada del programa

- `println!` es una macro (nota el `!`)
- Las cadenas de texto van entre comillas dobles

  Las macros en Rust terminan con `!`. Son diferentes de las funciones y se expanden en tiempo de compilación.

Compila y ejecuta con:

```
cargo run
```

Figure 1: Salida del primer programa en Rust

Variables y mutabilidad

En Rust, las variables son inmutables por defecto. Esta es una de las decisiones de diseño más importantes del lenguaje.

```
fn main() {
    let x = 5;
    println!("El valor de x es: {}", x);

    // Esto causaría un error:
    // x = 6;
}
```

Si necesitas mutar una variable, debes declararla explícitamente como mutable:

```
fn main() {
    let mut x = 5;
    println!("El valor de x es: {}", x);

    x = 6;
    println!("Ahora x es: {}", x);
}
```

Constantes vs Variables

Las constantes son diferentes de las variables inmutables:

```
const MAX_POINTS: u32 = 100_000;
```

Característica	Variable inmutable	Constante
`let`	✅	❌ usa `const`
Tipo explícito	Opcional	Obligatorio
Scope global	❌	✅
Valor en runtime	✅	❌ solo compile-time

Ownership

El concepto más importante y único de Rust es el ownership (propiedad). Este sistema es lo que permite a Rust garantizar seguridad de memoria sin un recolector de basura.

Las reglas del ownership

1. Cada valor en Rust tiene un dueño (owner)
2. Solo puede haber un dueño a la vez
3. Cuando el dueño sale del scope, el valor se elimina

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;  // s1 se "mueve" a s2

    // println!("{}", s1);  // ✗ Error: s1 ya no es válido
    println!("{}", s2);     // □ OK
}
```

El concepto de “move” en Rust es fundamental. A diferencia de otros lenguajes, cuando asignas una variable a otra, la primera deja de ser válida.

A Instalación de Rust

Aquí puedes incluir instrucciones detalladas de instalación para diferentes sistemas operativos.

B Recursos adicionales

Lista de libros, documentación, y comunidades recomendadas.

B.0.0.0.1 Implicaciones Prácticas

Debido a estas limitaciones de precisión:

1. **Nunca se deben comparar flotantes con ==** para verificar igualdad exacta:

```
let a = 0.1 + 0.2;
let b = 0.3;

// Incorrecto:
if a == b {  // Puede fallar debido a errores de redondeo
    println!("Son iguales");
}

// Correcto: comparar con epsilon de tolerancia
let epsilon = 1e-10;
if (a - b).abs() < epsilon {
    println!("Son aproximadamente iguales");
}
```

2. **Evitar operaciones que acumulen errores de redondeo** en bucles largos.
3. **Considerar tipos decimales exactos** para aplicaciones financieras donde la precisión decimal es crítica (crates como `rust_decimal`).

println!(“PI: {}”, std::f64::consts::PI); println!(“E (número de Euler): {}”, std::f64::consts::E); println! (“Raíz de 2: {}”, std::f64::consts::SQRT_2);

println!(“f32 epsilon: {}”, f32::EPSILON); println!(“f64 epsilon: {}”, f64::EPSILON);