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

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.
```
Output

Tu Primer Proyecto con Cargo

Crear un nuevo proyecto

```
# Crear un proyecto binario (aplicación)
cargo new a hola_mundo b
```
Terminal

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

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

Estructura creada:

```
hello_world/ Raíz del proyecto
├── 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
```
Output

Anatomía del Proyecto: Cargo.toml

Contenido inicial de Cargo.toml

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


[dependencies] crates
# Ejmplos
# rand = "0.8.5" Permite generar números aleatorios

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

(1) Metadatos

(2) Librerías externas

Punto de entrada: main.rs

```rust
fn main() A { C
    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

```terminal
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
```

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:

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

Variables con Rust

Let

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

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**

```output
Mi edad es 25
```

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);
}
```

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
  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.

```output
Valor de PI: 5.14
```

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);
}
```

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 (;).