Fases del Proceso

1. Paso 1: Creación del Módulo Fuente

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

   ```
   fn main() {
       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 hola_mundo
```

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

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

Estructura creada:

```
hello_world/
├── Cargo.lock
├── Cargo.toml
├── src/
│   └── main.rs
```

```
└── target/
    └── debug/
        ├── build/
        └── deps/
          └── hello_world
```

Anatomía del Proyecto: Cargo.toml

Contenido inicial de Cargo.toml

```
[package]
name = "hola_mundo"
version = "0.1.0"
edition = "2021"

[dependencies]
# Ejmplos
# rand = "0.8.5" Permite generar números aleatorios
# serde = "1.0.130" Permite serializar y deserializar datos

[dev-dependencies]
# Dependencias solo para desarrollo/testing

[build-dependencies]
# Dependencias para scripts de build
```

Punto de entrada: main.rs

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

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
```

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

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!
```

Variables con Rust

Let

```
fn main() {
  let edad = 25;
```

```
  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

```
Mi edad es 25
```

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

```
fn main() {
  let edad = 25;
  edad = 26;
  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`.
```

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.

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

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";
  println!("{}", edad);
}
```

- 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
```

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

```rust
let mut texto = "Hola";
texto = 5;
```

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

- 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
```

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: f64 = 3.14159265359;

fn main() {
  const PI: f64 = 5.14;
  println!("Valor de PI: {}", PI);
  //PI = 3.12;
}
```

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
```

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: u8 = 2;

fn main() {
  // let PROTOCOLO_VERSION:u8 = 3;
  // PROTOCOLO_VERSION: u8 = 8;
  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
```

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 = {
    let z = 3;
```

```
    z + 1
  };

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

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

     ```
     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

     ```
     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
```

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

1. A:

   Devuelve unit type ()

   ```
   y = ()
   ```

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.

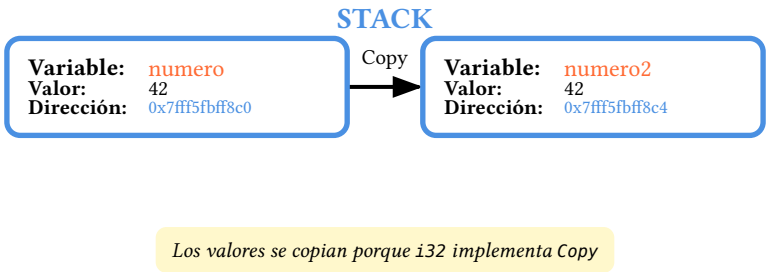
Los primeros conceptos pilares es comprender como se administra la memoria. Rust divide la memoria en dos grandes regiones Stack y Heap, su diferencia es esencial para evitar errores como:

- 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;
}
```



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

Diagrama 1: 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 = String::from("Rûst");
  let texto2: String = texto;

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

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.

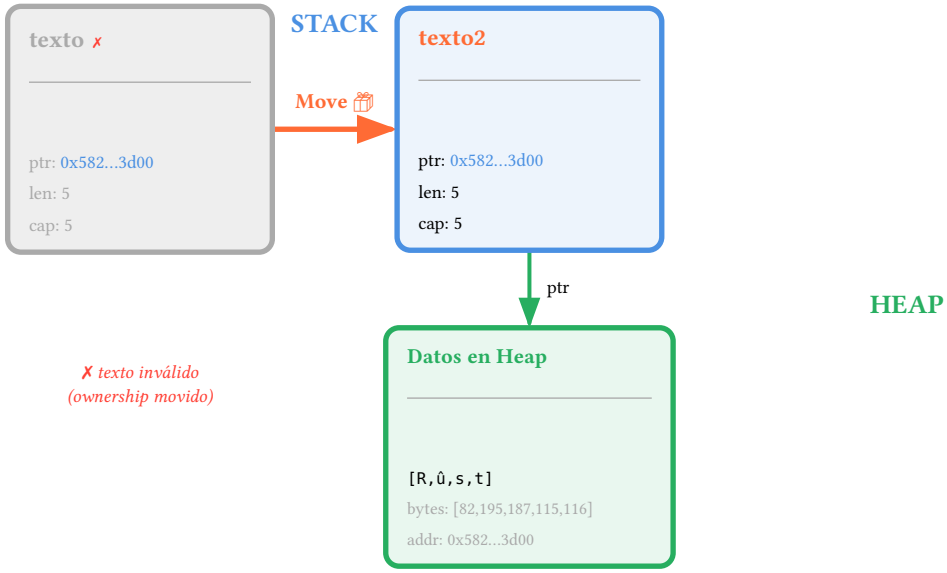


Diagrama 2: Move semántico: el ownership se transfiere de `texto` a `texto2`