Seminario de Lenguajes de Programación

Dayan Cabrera C311 Eveliz Espinaco C311 Michell Viu C311

7 de abril de 2025

1. Explique qué son Cargo y Crates.io y ejemplifique los beneficios de su uso

1.1 Cargo: El gestor de paquetes de Rust

¿Qué es?
Cargo es el sistema oficial de construcción y gestión de dependencias en Rust. Funciona como:

- Sistema de compilación (automáticamente llama a `rustc`)
- Gestor de dependencias (similar a `npm` en JavaScript o `pip` en Python)
- Herramienta de publicación de paquetes

Beneficios:

1. **Gestión automatizada de dependencias**
   Cargo resuelve automáticamente las versiones compatibles de librerías.
   Ejemplo (`Cargo.toml`):

```toml
[dependencies]
serde = "1.0"    # Versión mínima
tokio = { version = "1.0", features = ["full"] }
```

2. **Reproducibilidad garantizada**
   El archivo `Cargo.lock` congela versiones exactas.
   Ejemplo (salida de `Cargo.lock`):

```toml
[[package]]
name = "serde"
version = "1.0.198"  # Versión exacta usada
```

3. **Comandos unificados**
   Flujo de trabajo simplificado:

```bash
cargo new proyecto  # Crear proyecto
cargo build         # Compilar
cargo run           # Ejecutar
cargo test          # Probar
```

1.2 Crates.io: El repositorio de paquetes

¿Qué es?
Es el registro público centralizado donde:

- Los desarrolladores publican librerías ("crates")
- Cargo busca automáticamente las dependencias
- Se aplica control de versiones semántico (SemVer)

Beneficios:

1. **Acceso a ecosistema de calidad**
   Librerías populares como `serde` para serialización:

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
struct User {
    id: u32,
    name: String,
}
```

2. **Integración perfecta con Cargo**
   Añadir dependencias es trivial:

```toml
[dependencies]
reqwest = "0.11"  # Descargado automáticamente de Crates.io
```

3. **Seguridad y transparencia**
   Todas las crates son:
   - Open-source por diseño
   - Con historial de versiones público
   - Estadísticas de descargas visibles

   Ejemplo de estadísticas:
   `https://crates.io/crates/serde` muestra 100M+ descargas

2. Diferencia entre `name1::name2` y `name1.name2` en Rust

2.1 Operador `::` (Path Scope)

Uso principal: Acceder a elementos en el espacio de nombres (namespace).

- **Para módulos**:

```rust
// Acceder a una función dentro de un módulo
mod utilities {
    pub fn helper() { /* ... */ }
}

utilities::helper();  // Namespace path
```

- **Para tipos**:

```rust
// Acceder a un tipo dentro de un módulo
std::fs::File  // Tipo File del módulo fs en la librería std
```

- **Para asociados**:

```rust
// Acceder a constantes o funciones asociadas
f64::consts::PI  // Constante PI del módulo consts en el tipo f64
```

2.2 Operador `.` (Member Access)

Uso principal: Acceder a miembros de una instancia.

- **Campos de struct**:

```rust
struct User {
    name: String,
```

```
}

let user = User { name: String::from("Alice") };
println!("{}", user.name);  // Acceso a campo
```

- **Métodos**:

```
let nums = vec![1, 2, 3];
nums.push(4);  // Llamada a método de instancia
```

- **Trait objects**:

```
let text = "Rust";
text.to_uppercase();  // Método de trait
```

2.3 Diferencias Clave

Criterio	`::`	`.`
Contexto	Espacio de nombres	Instancia concreta
Uso	Tipos, módulos, funciones asociadas	Campos, métodos
Ejecución	Tiempo de compilación	Tiempo de ejecución
Ejemplo	`std::collections::HashMap`	`map.insert("key", "value")`

3. Manejo de Errores con `Result` y `Option`

3.1 El patrón `Result<T, E>`

Rust introduce el tipo `Result` como una **enum** para manejar operaciones que pueden fallar:

```
enum Result<T, E> {
    Ok(T),   // Valor correcto de tipo T
    Err(E),  // Error de tipo E
}
```

Beneficios:

- **Seguridad en tiempo de compilación**: Obliga a manejar ambos casos
- **Claridad semántica**: El tipo indica que la operación puede fallar
- **Flexibilidad**: Permite cualquier tipo para errores (`E`)

3.2 Uso práctico con funciones

Ejemplo de función que puede fallar:

```
fn dividir(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("División por cero"))
    } else {
        Ok(a / b)
    }
}
```

3.3 Manejo con `match`

El `match` es exhaustivo y obliga a manejar todos los casos:

```
match dividir(10.0, 2.0) {
    Ok(resultado) => println!("Resultado: {}", resultado),
    Err(e) => println!("Error: {}", e),
}
```

3.4 Generalización con Enums

`Result` es un caso especial de las **enums** en Rust. Otro ejemplo clave es `Option<T>`:

```
enum Option<T> {
    Some(T),  // Valor presente
    None,     // Valor ausente
}
```

Ejemplo combinado:

```
fn buscar_elemento(lista: &[i32], valor: i32) -> Option<usize> {
    lista.iter().position(&x x == valor)
}

fn procesar(lista: &[i32], valor: i32) -> Result<String, String> {
    match buscar_elemento(lista, valor) {
        Some(pos) => Ok(format!("Encontrado en posición {}", pos)),
        None => Err("Valor no encontrado".into()),
    }
}
```

3.5 Métodos útiles

Rust provee métodos para manejo conciso:

Método	Descripción	Ejemplo
`unwrap()`	Obtiene el valor o panic	`let x = Some(2).unwrap();`
`unwrap_or()`	Valor por defecto	`None.unwrap_or(0);`
`map()`	Transforma el valor interno	`Some(2).map(|x| x*2);`
`and_then()`	Encadenamiento	`Some(2).and_then(|x| Some(x*2));`
`?`	Propagación de errores	`let x = dividir(1.0, 0.0)?;`

4. Traits en Rust: Abstracción y Polimorfismo

4.1 ¿Qué son los Traits?

Los **traits** son una característica fundamental de Rust que permiten:

- Definir comportamientos compartidos entre tipos
- Establecer contratos de interfaz
- Habilitar polimorfismo sin herencia

Conceptualmente, son similares a las *interfaces* en otros lenguajes pero más potentes.

4.2 Sintaxis Básica

Definición de un trait simple:

```
trait Saludar {
    fn saludar(&self) -> String;  // Método abstracto

    fn saludar_fuerte(&self) -> String {  // Método con implementación por defecto
        format!("¡{}!", self.saludar())
    }
}
```

4.3 Implementación para Tipos

Implementación para un struct:

```
struct Persona {
    nombre: String,
}
```

```rust
impl Saludar for Persona {
    fn saludar(&self) -> String {
        format!("Hola, soy {}", self.nombre)
    }
    // saludar_fuerte usa la implementación por defecto
}
```

4.4 Uso Práctico

Ejemplo de llamada:

```rust
let ana = Persona { nombre: String::from("Ana") };
println!("{}", ana.saludar());         // "Hola, soy Ana"
println!("{}", ana.saludar_fuerte()); // "¡Hola, soy Ana!"
```

4.5 Traits como Parámetros

Polimorfismo mediante **trait bounds**:

```rust
fn imprimir_saludo<T: Saludar>(item: T) {
    println!("{}", item.saludar());
}

// Alternativa con sintaxis 'impl Trait'
fn imprimir_saludo_fuerte(item: &impl Saludar) {
    println!("{}", item.saludar_fuerte());
}
```

4.6 Traits Comunes de la Biblioteca Estándar

Trait	Propósito
`Debug`	Formateo para depuración (`{:?}`)
`Clone`	Clonación explícita de valores
`Drop`	Limpieza de recursos (similar a destructores)
`Iterator`	Trabajo con secuencias de elementos
`From/Into`	Conversiones entre tipos

4.7 Ejemplo Avanzado: Sistema de Notificaciones

```rust
trait Notificable {
    fn enviar(&self, mensaje: &str) -> bool;
}

struct Email {
    direccion: String,
}

impl Notificable for Email {
    fn enviar(&self, mensaje: &str) -> bool {
        println!("Enviando email a {}: {}", self.direccion, mensaje);
        true
    }
}

struct SMS {
    numero: String,
}

impl Notificable for SMS {
    fn enviar(&self, mensaje: &str) -> bool {
        println!("Enviando SMS a {}: {}", self.numero, mensaje);
        true
    }
}
```

```rust
fn notificar_masivo(servicios: Vec<&dyn Notificable>, mensaje: &str) {
    for servicio in servicios {
        servicio.enviar(mensaje);
    }
}
```

4.8 Tipos de Traits Especiales

- **Marker Traits**: Sin métodos (ej. `Send`, `Sync`)
- **Auto Traits**: Implementados automáticamente (ej. `Send`)
- **Object-Safe Traits**: Usables como `dyn Trait` (no pueden devolver `Self`)

4.9 Buenas Prácticas

- Preferir traits pequeños y específicos (como `Read + Write` vs `ReadWrite`)
- Usar `derive` para traits comunes (`Debug`, `Clone`, etc.)
- Evitar métodos con implementaciones por defecto a menos que sean lógicamente relacionados

---

```rust
#[derive(Debug, Clone, PartialEq)]
struct Punto {
    x: i32,
    y: i32,
}
```
