Manejo de Errores

Errores Recuperables

Rust trata el manejo de errores como una parte esencial del lenguaje.

Para evitar fallos inesperados, Rust propone una estrategia distinta a otros lenguajes: tratar los errores como valores, no como excepciones.

¿Qué es un error recuperable?

Un error recuperable es aquel que:

- Podemos anticipar
- Podemos manejar
- No requiere terminar el programa

Ejemplos:

- Un archivo que no existe
- Una conexión de red que falla
- Un número que no se puede convertir
- Un usuario que podría o no estar registrado

Rust usa `Option` y`Result` para estos casos.

Option<T>

valor o ausencia de valor

```
enum Option<T> {
    Some(T),
    None,
}
```

Sirve para modelar cosas que pueden o no existir.

Obtener el primer elemento de un vector

```
fn obtener_primero(nums: &Vec<i32>) -> Option<i32> {
    nums.first().copied()
}
```

Uso:

```
match obtener_primero(&vec![10, 20, 30]) {
    Some(n) => println!("Primer número: {}", n),
    None => println!("El vector está vacío"),
}
```

Métodos importantes de Option<T>

- .unwrap()

  Obtiene el valor o hace panic
- .unwrap_or(x)

  Valor por defecto
- .unwrap_or_else()

  Llama una función para obtener un valor

- .map()

  Transforma Some
- .and_then()

  Encadena operaciones que también devuelven Option

Ejemplo: evitar panic usando unwrap_or

```
let nombre = Some("Juan");
let n = nombre.unwrap_or("Invitado");
```

Result<T, E>
operación que puede fallar

Definición:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Ejemplos: abrir archivos, parsear strings, convertir tipos.

Convertir un string en número

```
fn convertir(s: &str) -> Result<i32, std::num::ParseIntError> {
    s.parse::<i32>()
}
```

Uso:

```
match convertir("42") {
    Ok(n) => println!("Número: {}", n),
    Err(e) => println!("Error: {}", e),
}
```

Métodos importantes de Result<T, E>

- .unwrap()

  Obtiene T o panic
- .expect(msg)

  Igual que unwrap pero con mensaje
- .map()

  Transforma Ok
- .map_err()

  Transforma Err
- .ok()

  Convierte Result a Option
- .err()

  Obtiene la parte Err

El operador ?
El operador ?:

- Si es Ok, devuelve el valor

- Si es Err, retorna inmediatamente de la función

Ejemplo sin ?

```
fn leer_archivo() -> Result<String, std::io::Error> {
    let mut archivo = File::open("hola.txt");
    match archivo {
        Ok(mut f) => {
            let mut contenido = String::new();
            match f.read_to_string(&mut contenido) {
                Ok(_) => Ok(contenido),
                Err(e) => Err(e),
            }
        }
        Err(e) => Err(e),
    }
}
```

Ejemplo con ?

```
fn leer_archivo() -> Result<String, std::io::Error> {
    let mut contenido = String::new();
    File::open("hola.txt")?.read_to_string(&mut contenido)?;
    Ok(contenido)
}
```

Patrones para manejar errores correctamente

match

```
match archivo {
    Ok(a) => println!("Abierto"),
    Err(e) => println!("Error: {}", e),
}
```

if let

```
if let Some(n) = x {
    println!("El número es {}", n);
}
```

while let

Ideal para iteradores:

```
let mut iter = vec![1,2,3].into_iter();
while let Some(v) = iter.next() {
    println!("{}", v);
}
```

Errores NO Recuperables

¿Qué es un error no recuperable?

Un error no recuperable es cuando la única opción segura es detener el programa inmediatamente para evitar corrupción de datos o comportamiento impredecible.

Ejemplos:

- Índice fuera de rango en un vector
- Dividir entre cero

- Usar unwrap() sobre None o Err
- Violaciones lógicas del código (invariantes imposibles)

Rust usa panic! para estos casos.

panic!
Detener el programa de forma controlada

panic! es una macro que termina la ejecución y muestra un mensaje.

```
fn main() {
    panic!("Ocurrió un error inesperado");
}
```

Salida típica:

```
thread 'main' panicked at 'Ocurrió un error inesperado', src/main.rs:2:5
```

¿Cuándo ocurre un panic automáticamente?
1. Índices fuera de rango

```
let v = vec![1, 2, 3];
println!("{}", v[10]); // panic!
```

1. unwrap() sobre None

```
let x: Option<i32> = None;
x.unwrap(); // panic
```

1. expect() también provoca panic

```
let r: Result<i32, _> = "hola".parse();
r.expect("No se pudo convertir"); // panic con mensaje personalizado
```

Cuándo no usar unwrap() y expect()

Ambos métodos causan panic si:

- Option es None
- Result es Err

¿Por qué existen entonces?

Porque en prototipos, scripts rápidos y tests es conveniente usarlos.

Cuándo evitar unwrap/expect:

- En producción
- En código que maneja errores del usuario
- En operaciones de I/O, red, archivos
- Cuando el valor puede no existir o fallar

Backtrace
seguir el origen del panic

Rust puede mostrar la ruta de llamadas que llevó al error.

Para activarlo:

```
RUST_BACKTRACE=1 cargo run
```

Ejemplo de salida:

```
thread 'main' panicked at 'fallo X', src/main.rs:12:9
stack backtrace:
   0: rust_begin_unwind
   1: core::panicking::panic_fmt
   2: my_project::funcion
   3: my_project::main
```

Esto es útil para depurar.