Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Input/Output. Result<A, E>

Input/Output

Ввід-вивід (I/O) у мовах програмування – це механізм обміну даними **між програмою та зовнішнім середовищем**, таким як користувач, файлові системи, мережеві з'єднання або інші пристрої. Вхідні (input) операції дозволяють отримувати дані для обробки, наприклад, з клавіатури, файлу або мережевого запиту, тоді як вихідні (output) операції забезпечують передачу результатів у вигляді тексту на екран, запису у файл чи надсилання по мережі. У більшості мов програмування для цього використовуються спеціальні функції або бібліотеки.

Input/Output

Ввід-вивід буває **синхронними** або **асинхронними**. Синхронний ввід-вивід передбачає, що програма чекає завершення операції перед виконанням наступних команд, що може сповільнювати роботу. Натомість асинхронний підхід дозволяє програмі продовжувати виконання, поки операція вводу-виводу ще триває, що підвищує продуктивність, особливо у багатопотокових і мережевих програмах. У сучасних мовах програмування широко використовуються буферизація, потоки (streams) та неблокуючі операції для ефективного керування ввід-виводом.

Input

- клавіатура
- файл
- база даних
- порт операційної системи (залізо)
- API
- ...

Output

- консоль (екран)
- файл
- база даних
- порт операційної системи (залізо)
- API
- ...

Output детальніше. Консоль

```
println!("x = 5");
println!("{}", "error".red());
```

```
x = 5
error
```

https://doc.rust-lang.org/rust-by-example/hello/print.html

Output детальніше. Консоль

```
///                                                    0      1
println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");
```

```
Alice, this is Bob. Bob, this is Alice
```

Output. Файл

Текстовий файл

```
let mut file : File = File::create( path: "output.txt")?;
writeln!(file, "Це запис у текстовий файл!")?;
```

Output. Файл

Бінарний файл

```
let file : File = File::create( path: "output.bin")?;
let mut file : BufWriter<File> = BufWriter::new(file);
file.write_all( buf: &[0x42, 0x55, 0x53, 0x59])?;
```

Output. Файл

Бінарний файл

```
let file : File = File::create( path: "output.bin")?;
let mut file : BufWriter<File> = BufWriter::new(file);
file.write_all( buf: &[0x42, 0x55, 0x53, 0x59])?;
```

Output. Файл

```
let file_r : Result<File> = File::create( path: "output.bin");
match file_r {
    Err(e : Error ) => println!("error during file create {e}"),
    Ok(file : File ) => {
        let mut buf : BufWriter<File> = BufWriter::new(file);
        let r : Result<()> = buf.write_all( buf: &[0x42, 0x55, 0x53, 0x59]);
        match r {
            Err(e : Error ) => println!("error during file write {e}"),
            Ok(_) => println!("file wrote without errors"),
        }
    }
}
```

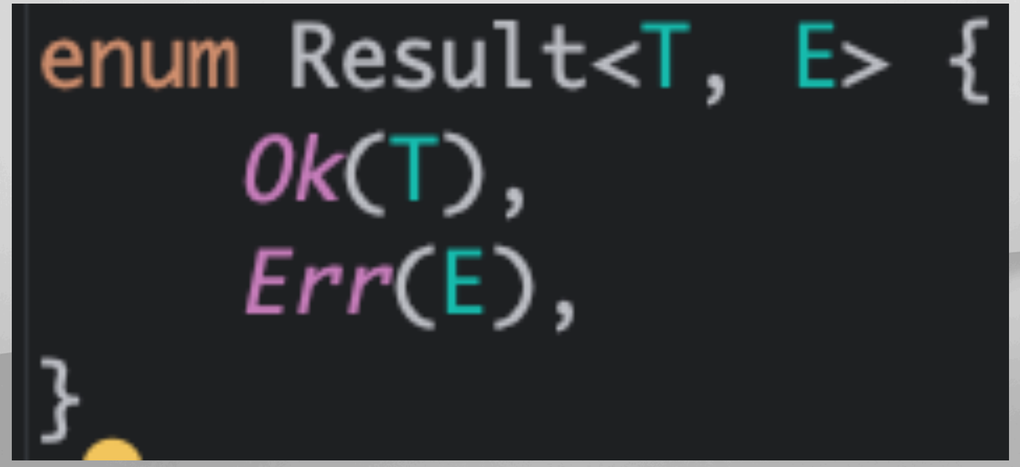
Output. Файл

Але синтаксис мови програмування Rust дозволяє використовувати синтаксис ?
І це дозволяє автоматично працювати тільки з ситуаціями коли нема помилок (Ok) а помилка (Err) автоматично “протікає” з функції, але в цьому випадку треба змінити сигнатуру функції

```rust
#[test]
fn test2_write_bin_file() -> std::io::Result<()> {
    let file: File = File::create(path: "output.bin")?;
    let mut buf: BufWriter<File> = BufWriter::new(file);
    buf.write_all(buf: &[0x42, 0x55, 0x53, 0x59])?;
    Ok(())
}
```

Input/Output. Файл

Технічно будь який I/O може “зламатися” тому всі операції I/O повертають Result<A>, який на справді є Result<A, E>
Або результат типу **A**
Або помилка типу **E**



Input/Output. Http запит

```rust
async fn f1() -> Result<(), reqwest::Error> {
    let response : String = reqwest::get( url: "https://api64.ipify.org?format=text")
        .await? : Response
        .text() : impl Future<Output=Result<...>>+Sized
        .await?;
    println!("Ваш IP-адрес: {}", response);
    Ok(())
}
```

```
Ваша IP-адреса: 185.155.90.88
```

Input/Output. Hardware

```rust
let mut sys: System = System::new_all();
sys.refresh_all();

for cpu: &Cpu in sys.cpus() {
    println!("Brand: {}, Freq: {}", cpu.brand(), cpu.frequency(),);
}
```

```
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
Brand: Intel(R) Core(TM) i7-8700B CPU @ 3.20GHz, Freq: 3200
```

Input/Output. SQL Database

```rust
let cred : &str = "host=localhost user=postgres password=pg123456 dbname=fs8";
let (client : Client, connection : Connection<Socket, NoTlsStream> ) =
    tokio_postgres::connect(cred, NoTls).await?;

tokio::spawn( future: async move {
    if let Err(e : Error ) = connection.await {
        eprintln!("Помилка підключення: {}", e);
    }
});

let stmt : &str = "INSERT INTO person (name, salary) VALUES ($1, $2)";

client
    .execute(stmt,  params: &[&"Олексій", &5000]) : impl Future<Output=Result<...>>+Sized
    .await?;
```

Input детальніше. Консоль (Клавіатура)

```
print!("Enter your name:");

let mut buffer : String = String::new();

io::stdin()
    .read_line(&mut buffer) : Result<usize>
    .expect( msg: "Failed to read line");

println!("You entered: {}", buffer);
```

Input. Файл.

```
let contents : String = fs::read_to_string( path: "file.txt")?;
```

За допомогою цієї функції можливо відразу прочитати весь файл і покласти його вміст у String

Але у цього підходу є серйозний мінус.
Якщо файл дуже великий. Це буде тривати дуже довго,
та й взагалі може скінчитися пам’ять

Input. Файл.

```rust
let file : File = File::open( path: "input.txt")?;
let reader : BufReader<File> = io::BufReader::new(file);

for line : Result<String> in reader.lines() {
    let line : String = line?;
    println!("Read: {}", line);
}
```

Цей підхід принципово інший. Створюється ітератор і з кожним викликом читається один рядок. Набагато швидше. Дані доступні відразу після читання першого рядку, але є нюанс:
Неможливо дістатися попередньої або наступної строки.

Result<A, E>

Скоріш за все ви побачили, що в Rust більшість функцій input/output повертають не значення а Result<A, E>.

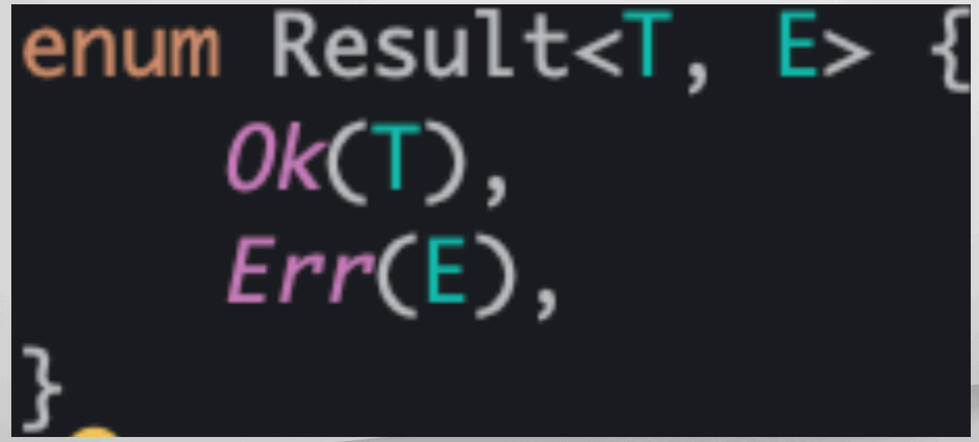
```
let mut buffer : String = String::new();
let size: Result<usize, Error> =
    io::stdin().read_line(&mut buffer);

let file: Result<File, Error> =
    File::open( path: "input.txt");

let sql: Result<(Client, Connection<Socket, NoTlsStream>), tokio_postgres::Error> =
    tokio_postgres::connect( config: "...", NoTls).await;
```

Result<A, E>

Це тому що в Rust нема exception і результатом I/O операції завжди є або результат, або помилка.
Для цієї цілі в Rust є тип (enum) Result<A, E>



Або результат типу **A**
Або помилка типу **E**

1. I/O є ненадійною операцією

Робота з файлами, мережею чи стандартним введенням/виведенням може не гарантувати успіх.

Файл може не існувати, бути заблокованим або недоступним через права доступу.

При роботі з мережею можуть виникнути відключення, тайм-аути чи помилки DNS.

Rust не дозволяє ігнорувати ці потенційні проблеми.

2. Явна обробка помилок – менше неочікуваних збоїв

У мовах, де використовується виняткова обробка (try-catch), помилки можуть бути непередбачуваними та не оброблятися правильно.

У Rust Result<A, E> змушує програміста явно опрацьовувати можливі збої, зменшуючи ризик аварійного завершення програми.

3. Відсутність механізму винятків (exceptions)
Rust не використовує винятки (try-catch), бо вони ускладнюють контроль потоку виконання.
Замість цього він застосовує Result<T, E>, що дозволяє безпечно та передбачувано обробляти помилки.

4. Безпечне керування ресурсами
У Result<A, E> тип E вказує на конкретну помилку, що допомагає аналізувати причину збою без втрати інформації.
Це уникає ситуацій, коли помилки губляться через глобальні винятки.

Result<A, E>

Основній шаблон використання Result<A, E> це
1. Якщо нас “не цікавлять” помилки, то можемо їх ігнорувати:

```
let mut buffer :String = String::new();
let size :usize = io::stdin().read_line(&mut buffer).unwrap();

let file :File = File::open( path: "input.txt").unwrap();
```

Але магії нема, і якщо помилка трапиться, то це призведе до завершення програми з panic, бо трапилася ситуація яку ми не “описали” в сенсі “що з нею робити”

Result<A, E>

2. Постійно перевіряти результат і “якісно” опрацьовувати помилку. Тобто на кожному кроці “описувати” поведінку програми “якщо помилка”.

```
let mut file : Result<File> = File::create( path: "output.txt");
match file {
    Ok(mut file : File ) => {
        let x : Result<()> = writeln!(file, "Це запис у текстовий файл!");
        match x {
            Ok(_) => {
                println!("file wrote good");
                Ok(())
            }
            Err(e : Error ) => {
                println!("error during file write {e}");
                Err(e)
            }
        }
```

Result<A, E>

Але існують інші “комбінатори”

```
fn test3(s: String) {
    let x : Result<i32, ParseIntError> = s.parse::<i32>();
    let k : Result<i32, ParseIntError> = x.map(|n : i32 | n * 10);
    let z : i32 = k.unwrap_or( default: -13);
}
```

Result можна розглядати, як “коробку” з якою можна маніпулювати за допомогою методу **.map**

Result<A, E>

.map(f) просто вже має реалізацію

```
pub fn map<U, F: FnOnce(T) -> U>(self, op: F) -> Result<U, E>
    match self {
        Ok(t : T ) => Ok(op(t)),
        Err(e : E ) => Err(e),
    }
}
```

і дозволяє писати набагато менше коду.

Result<A, E>

Таких “комбінаторів” не 2 і не 3, їх ~100
Але цікавість у тому, що всі вони приймають параметром
або функцію **A => B,**
або функцію **E => E2,**
або значення типу **A**
бо кінцева ідея або “дістати” значення типу A
Або запустити різний код для 2х можливих варіантів.

Висновки

Механізм вводу-виводу є одним із ключових аспектів будь-якої мови програмування, оскільки він дозволяє програмам не лише взаємодіяти з користувачем, а й працювати з файлами, базами даних, мережею та апаратним забезпеченням. Без ефективного I/O програмне забезпечення залишалося б ізольованим, обмеженим лише внутрішньою логікою. Завдяки засобам вводу-виводу можна реалізовувати складні системи, що зчитують дані з різних джерел, обробляють їх і передають результати у потрібному форматі.

Висновки

Операції вводу-виводу також відіграють важливу роль у розширенні можливостей інтеграції між різними програмними компонентами. Наприклад, через роботу з мережевими API програми можуть взаємодіяти з веб-сервісами, отримуючи актуальні дані або передаючи результати обчислень на віддалені сервери. Використання баз даних дозволяє зберігати та обробляти великі обсяги інформації, а взаємодія з апаратним забезпеченням відкриває можливість розробки вбудованих систем, моніторингу стану пристроїв або управління ними.

Висновки

Загалом, механізми вводу-виводу є основою для створення гнучких та масштабованих систем, які можуть працювати в різних середовищах і обмінюватися даними між численними компонентами. Вони дозволяють реалізовувати ефективну комунікацію між користувачем, програмним забезпеченням та апаратними ресурсами, що робить мови програмування універсальними інструментами для вирішення широкого спектра завдань.

Код з лекцій,
презентації Keynote,
PDF-файли
знаходяться на GitHub:

https://github.com/djnzx/rust-course
git@github.com:djnzx/rust-course.git