Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Functions. Closures

Чому...

- **маючи гроші**,
  ми все одно не закінчуємо проекти вчасно

- **маючи досвід**,
  ми все одно не знаємо коли буде готовий проєкт

- **маючи найкращі інструменти**,
  ми все одно пишемо поганий код

- **...**
  ...

Тому що...

Структура

Відділення коду від даних

Відділення коду від даних є важливим аспектом, який забезпечує гнучкість і масштабованість програм. Замість того щоб дані були жорстко закріплені в коді, їх зберігають у конфігураційних файлах, базах даних чи інших зовнішніх джерелах. Це дозволяє легко змінювати поведінку програми без необхідності модифікації основного коду. Крім того, відділення логіки від даних забезпечує кращу підтримку та розширюваність системи, адже зміни у даних не впливають на логіку і навпаки. Програмісти можуть зосередитися на оптимізації обробки даних, не турбуючись про зміну їхніх значень чи джерел.

Поділ на функції

Поділ коду на функції – один з фундаментальних принципів структурного програмування. Він дозволяє розбивати програму на менші, незалежні компоненти, кожен з яких виконує конкретне завдання. Це забезпечує повторне використання коду та покращує його тестованість. Декомпозуючи програму на невеликі функції, програмісти отримують можливість поліпшити структуру коду та зробити його більш читабельним. Крім того, завдяки такій структурі полегшується пошук і виправлення помилок, адже кожна функція відповідає лише за одну конкретну операцію.

Принцип єдиної відповідальності

Принцип єдиної відповідальності стверджує, що кожен модуль чи клас повинен мати лише одну відповідальність, тобто виконувати лише одну логічно завершену задачу. Це робить код більш зрозумілим, легким для підтримки і модифікації. Наприклад, якщо в одному класі, модулі, або функції одночасно реалізовано логіку для обробки даних та їх відображення, це може призвести до ускладнень у разі зміни будь-якої з цих частин. Виділяючи окремі відповідальності у різні класи, можна легше вносити зміни, не ризикуючи порушити роботу інших частин програми.

Групування коду.

Групування коду за функціональним призначенням є критичним для забезпечення зрозумілості та підтримуваності програмного забезпечення. Коли всі частини коду, що відповідають за певну задачу або функціональність, згруповані разом, це значно полегшує роботу над програмою як для поточних розробників, так і для майбутніх команд. Така організація дозволяє швидко знайти й змінити потрібну функціональність, не втрачаючи часу на пошук розкиданих частин логіки по всій системі. Чітке групування також спрощує тестування, оскільки кожен модуль можна перевіряти окремо.

Функція Базовий синтаксис

```rust
fn add(a: i32, b: i32) -> i32 {
    let c :i32 = a + b;
    return c;
}
```

```rust
fn add(a: i32, b: i32) -> i32 {
    let c :i32 = a + b;
    c
}
```

Чому ми завжди пишемо?

```
fn launch(a: A) -> B;
```

Навіть коли

```
fn add(a: i32, b: f64) -> f64;
fn div(a: i32, b: i32) -> (i32, i32);
fn print(s: String);
fn get_a_number() -> i32;
fn destroy_universe();
```

Тому що

```
A = (i32, i32)
```

```
A = ()
```

```
B = (String, i32)
```

```
B = ()
```

```
...
```

І відповідно

```
fn add1(a: i32, b: f64) -> f64;

fn add2(ab: (i32, f64)) -> f64;
```

І також

```rust
fn get_a_number1(_x: ()) -> i32;

fn get_a_number2() -> i32;
```

І також

```rust
fn destroy_universe1() -> ();

fn destroy_universe2();
```

І будь які комбінації…

Що таке () або Unit?

У мові програмування **Rust** тип **()** (називається **unit**) є спеціальним типом, що представляє значення **"нічого"**. Його семантика та значення подібні до інших мов програмування, таких як **void** у C або Java. Проте в Rust тип **()** використовується більш виразно і має певні особливості.

Семантика типу () Unit.

Порожнє значення: Значення типу **()** вказує на те, що певна функція або вираз не повертає значущого значення. Це використовується для тих функцій, які виконують дії, але не мають результату, наприклад, функції типу "процедура" або функції з побічними ефектами (як, наприклад, виведення на екран).

Тип **()** має **лише одне можливе значення** – порожні круглі дужки (). Це єдине значення для цього типу, і воно несе в собі лише семантичне значення "нічого".

Тип Unit вказувати не обов’язково

```rust
fn returns_unit1(x: i32) -> () {
    println!("hello");
}

fn returns_unit2(x: i32) {
    println!("hello");
}
```

Але...

```rust
fn inc1(x: i32) -> () {
    x + 1;
}

fn inc2(x: i32) {
    x + 1;
}

fn inc(x: i32) -> i32 {
    x + 1
}
```

Приклади коду, повертаючого Unit

```rust
fn returns_unit2(x: i32) {
    println!("hello");
}

fn returns_unit3(mut x: i32) {
    x += 1;
}

fn returns_unit4(mut x: i32) {
    let x :() = for i :i32 in 0..10 {
        // whatever
    };
}
```

Вкладений Scope

```
//
let x :i32 = {
    let a :i32 = 1 + 2;
    let b :i32 = 3 + 4;
    a + b
};
let c = a + b;
//
```

Вкладені функції

- **Інкапсуляція логіки**: Вкладені функції дозволяють інкапсулювати певну частину логіки всередині іншої функції, роблячи її недоступною зовні. Це підвищує безпеку коду і зменшує ризик ненавмисних змін або помилок, оскільки вкладена функція є локальною і може використовуватися лише в межах зовнішньої функції.
- **Організація коду**: Вкладені функції допомагають організувати код, роблячи його більш структурованим і зрозумілим. Це дозволяє розбити складну функцію на менші логічні частини, кожна з яких виконує окрему задачу. Це також полегшує читабельність і підтримку, оскільки логіка, що стосується конкретної частини завдання, згрупована в одному місці.
- **Узгодженість з принципом єдиної відповідальності**: Використання вкладених функцій дозволяє дотримуватися принципу єдиної відповідальності, де кожна функція виконує свою чітку задачу. Вкладені функції можуть використовуватися для винесення допоміжної логіки, що не є основною для зовнішньої функції, але є важливою частиною її реалізації.

Вкладені функції. Синтаксис

```rust
fn solve(x: i32, y: i32) -> i32 {

    fn solve1(a: i32, b: i32) -> i32 { todo!() }

    fn solve2(a: i32, b: i32) -> i32 { todo!() }

    solve1(x, y) + solve2(x, y)
}
```

Вкладені функції не мають доступу до параметрів основної функції

```
fn solve2(x: i32, y: i32) -> i32 {

    fn solve_part(a: i32, b: i32) -> i32 {
        a + b + x
    }

    todo!()
}
```

Але інколи доступ потрібен

Приклад

```rust
fn align_all(data: &[&str], width: u8) -> Vec<String> {...}

#[test]
fn test2() {
    align_all( data: &["abc", "ab", "a", "abcdef"], width: 10)
        .iter() : impl Iterator<Item=&String>
        .for_each(|x : &String | println!(">{}<", x));
}
// >abc        <
// >ab         <
// >a          <
// >abcdef     <
```

Приклад

```
fn align_all(data: &[&str], width: u8) -> Vec<String> {

    fn align(s: &str, width: u8) -> String {
        s.to_string() + &*" ".repeat( n: width as usize - s.len())
    }

    data.iter() : impl Iterator<Item=&&str>
        .map(|x : &&str | align(x, width)) : impl Iterator<Item=String>
        .collect::<Vec<String>>()
}
```

Вирішення проблеми. Closures (Замикання)

```rust
fn align_all_v2(data: &[&str], width: u8) -> Vec<String> {

    let align : fn(&str) → String =
        |s: &str|
            s.to_string() + &*" ".repeat( n: width as usize - s.len());

    data.iter() : impl Iterator<Item=&&str>
        .map(|x : &&str | align(x)) : impl Iterator<Item=String>
        .collect::<Vec<String>>()
}
```

Closures. Синтаксис

```rust
fn closures_syntax(delta: i32) {

    let with_delta : fn(i32) -> i32 = |x: i32| x + delta;

    let total : i32 = with_delta(100);
    println!("delta was: {delta}");
    println!("total (100+{delta}): {total}");
}
```

Closures. Синтаксис

```rust
#[test]
fn test3() {
    closures_syntax( delta: 1);
}
// delta was: 1
// total (100+1): 101
```

Дотримання принципів, таких як Single Responsibility, поділ на функції, відділення коду від даних та правильне структурування програм, є критично важливими для створення якісного програмного забезпечення.

Вони допомагають зменшити складність системи, роблять її більш модульною і легшою для розуміння.

Це особливо важливо для довготривалих проєктів, де підтримка і розширення коду можуть займати більше часу, ніж початковий його написання. Модульність і чітке розмежування відповідальностей забезпечують легкість у внесенні змін, тестуванні та відлагодженні.

Крім того, такі підходи дозволяють створювати код, який є більш гнучким та адаптивним до змін. Оскільки вимоги до програмного забезпечення постійно еволюціонують, важливо, щоб система могла легко адаптуватися без потреби в повному її переписуванні. Це економить час і ресурси, а також знижує ризики виникнення помилок у процесі розробки й обслуговування.

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

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