Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Pattern Matching

Складні типи

Tuple,
Named Tuple,
Struct,
Enum
If .. Else …

З цим треба щось робити

Pattern matching

Це одна з ключових особливостей мови Rust, яка дозволяє визначати контроль за виконанням програми на основі структури даних. Це потужний інструмент для роботи зі складними структурами та даними, що дозволяє зручно розбивати їх на компоненти та приймати рішення залежно від їх змісту.

На відміну від інших мов програмування, де контроль за виконанням часто реалізується через умовні оператори (**if-else** або **switch**), Rust надає розширену підтримку через конструкції **match**, **if let**, та інші.

Навіщо?

Патерн-матчинг дозволяє:

- Ефективно перевіряти та обробляти різні типи даних.
- Гарантувати повноту перевірки, оскільки Rust примушує розглядати всі можливі варіанти.
- Писати чистіший та більш читабельний код.

Приклади

Примітивні типи. Проблема ???

```rust
fn problem01(x: i32) {
    if x == 1 {
        println!("one")
    } else if x == 2 {
        println!("two")
    } else if x == -2 {
        println!("minus two")
    }
}
```

Складність коду

Примітивні типи. Вирішення

```rust
fn solution01(x: i32) {
    match x {
        1 => println!("one"),
        2 => println!("two"),
        -2 => println!("minus two"),
    }
}
```

Примітивні типи. Вирішення

```rust
fn solution02(x: bool) {
    match x {
        true => println!("t"),
        false => println!("f"),
    }
}
```

Примітивні типи. Комбінації. Зручний синтаксис для || (logical OR)

```rust
fn solution03(number: i32) {
    match number {
        1 => println!("One!"),
        2 | 3 | 5 | 7 | 11 => println!("This is a prime number!"),
    }
}
```

Примітивні типи. Комбінації. Зручний синтаксис для a..b (ranges)

```rust
fn solution04(number: i32) {
    match number {
        1 => println!("One!"),
        10..=19 => println!("A 10+ number!"),
    }
}
```

Зручний синтаксис для else

```rust
fn solution05(number: i32) {
    match number {
        1 => println!("One!"),
        _ => println!("Something else!"),
    }
}
```

Довільні комбінації

```
let z :&str = match x {
    1 => "One",
    2 | 3 => "Two or Three",
    4..10 => "4 till 10",
    _ => "10 or more",
};
```

Зручний синтаксис для Arrays

```rust
fn arrays01(xs: &[i32]) {
    match xs {
        [] => println!("array is empty"),
        [x :&i32] => println!("array has one element: {x}"),
        [x :&i32, y :&i32] => println!("array has two elements: {x}, {y}"),
        [13, ..] => println!("array starts from number 13: {xs:?}"),
        [7, 13, ..] => println!("array starts from numbers 7, 13: {xs:?}"),
        [.., 11, 53] => println!("array ends with numbers 11, 53 {xs:?}"),
        [.., 53] => println!("array ends with number 53 {xs:?}"),
        [1, .., 11] => println!("array starts from 1 ends with 11 {xs:?}"),
        _ => println!("something different {xs:?}"),
    }
}
```

Зручний синтаксис для Products Tuples

```rust
fn test(xy: (i32, i32)) {
    let s :String = match xy {
        (0, y :i32) => format!("on the axis X, y={y}"),
        (x :i32, 0) => format!("on the axis Y, x={x}"),
        (1, 1) => "at x=1, y=0".to_owned(),
        (x :i32, y :i32) => format!("x={x}, y={y}")
    };
}
```

Зручний синтаксис для Products
Named Tuples

```
struct Point(i32, i32);

fn test(xy: Point) {
    let s :String = match xy {
        Point(0, y :i32) => format!("on the axis X, y={y}"),
        Point(x :i32, 0) => format!("on the axis Y, x={x}"),
        Point(1, 1) => "at x=1, y=0".to_owned(),
        Point(x :i32, y :i32) => format!("x={x}, y={y}"),
    };
}
```

Зручний синтаксис для Products Structs

```
struct Pizza {
    name: String,
    size: u8,
}
```

```
fn test(p: Pizza) {
    match p {
        Pizza { name: String, size: u8 }
            if name == "Margarita" =>
                println!("Margarita of size {}", size),
        Pizza { name: String, size: 30 } =>
            println!("Pizza of size 30 with name: {name}",),
        Pizza { name: String, size: u8 } =>
            println!("pizza name: {}, size: {}", name, size),
    };
}
```

Зручний синтаксис для CoProducts

```
enum Solution {
    NoRoots,
    OneRoot(f32),
    TwoRoots(f32, f32),
}

fn solve_quadratic(a: f32, b: f32, c: f32) -> Solution
```

```
match solve_quadratic(a, b, c) {
    NoRoots =>
        println!("quadratic equation has no roots"),
    OneRoot(x :f32 ) =>
        println!("quadratic equation has one root: {x}"),
    TwoRoots(x1 :f32 , x2 :f32 ) =>
        println!("quadratic equation has two roots: {x1}, {x2}"),
}
```

Зручний синтаксис для extra if. Було

```
let d :f32 = b.powi( n: 2) - 4.0 * a * c;

if d < 0. {
    NoRoots
} else if d == 0. {
    OneRoot(-b / (2. * a))
} else {
    let dq :f32 = f32::sqrt(d);
    let a2 :f32 = a * 2.;
    TwoRoots((-b - dq) / a2, (-b + dq) / a2)
}
```

Зручний синтаксис для extra if. Стало

```
match d {
    d :f32 if d > 0. => {
        let dq :f32 = f32::sqrt(d);
        let a2 :f32 = a * 2.;
        TwoRoots((-b - dq) / a2, (-b + dq) / a2)
    }
    0. => OneRoot(-b / (2. * a)),
    _ => NoRoots,
}
```

macro!

```rust
fn code2() {
    let alpha : [char; 7] = ['a', 'E', 'Z', '0', 'x', '9', 'y'];
    for ab : char in alpha {
        /// macro to generate match statements
        let m : bool = matches!(ab, 'A'..='Z' | 'a'..='z' | '0'..='9');
        assert!(m);
    }
}
```

Автоматично конвертується

```rust
let m : bool = match ab {
    'A'..='Z' => true,
    'a'..='z' => true,
    '0'..='9' => true,
    _ => false
};
```

Exhaustiveness check. Enums

```
match solve_quadratic(a, b, c) {
    OneRoot(x :f32 ) =>
        println!("quadratic equation has one root: {x}"),
    TwoRoots(x1 :f32 , x2 :f32 ) =>
        println!("quadratic equation has two roots: {x1}, {x2}"),
}
```

Will not compile

```
enum Solution {
    NoRoots,
    OneRoot(f32),
    TwoRoots(f32, f32),
}
```

Exhaustiveness check. Any Product

```
fn test(xy: (i32, i32)) {
    let s :String = match xy {
        (0, y :i32) => format!("on the axis X, y={y}"),
        (x :i32, 0) => format!("on the axis Y, x={x}"),
        (1, 1) => "at x=1, y=0".to_owned(),
    };
}
```

Will not compile

Exhaustiveness check. Primitives

```
match x {
    1 => println!("one"),
    2 => println!("two"),
}
```

Will not compile

Exhaustiveness check. Any

```
for y:u32 in 1..=H {
    for x:u32 in 1..=W {
        let c: char = match (x, y) {
            (_, 1) => '-',
            (_, H) => '-',
            (1, _) => '|',
            (W, _) => '|',
            _ if x == mul(y, k) => '\\',
            _ if x == W - mul(y, k) => '/',
        };
```

Will not compile

Exhaustiveness check. Any Combination

```
let c: char = match (x, y) {
    (_, 1 | H) => '-',
    (1 | W, _) => '|',
    _ if x == mul(y, k) => '\\',
    _ if x == W - mul(y, k) => '/',
    _ => ' ',
};
```

Will not compile

Висновок

Патерн-матчинг дозволяє:

Патерн-матчинг у Rust є потужним і зручним інструментом для роботи з різними типами даних. Його використання дозволяє написати **чистий**, ефективний та **безпечний** код. Навички роботи з патерн-матчингом відкривають двері до більшої гнучкості у вирішенні складних завдань, таких як обробка результатів, помилок та складних структур даних.

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

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