Структури даних у Rust

Масиви • Кортежі • Option • match • Слайси • Структури

Лекція 2

Бичков Олексій Сергійович
Кафедра теоретичної кібернетики

Зміст лекції

1. Масиви [T; N] — фіксований розмір, один тип
2. Option<T> — обробка відсутніх значень (Some/None)
3. match — pattern matching (зіставлення з шаблоном)
4. Кортежі (T1, T2) — групування різних типів
5. Посилання &T — borrowing (запозичення)
6. Слайси &[T] — "вікно" в дані, fat pointer
7. Структури struct — власні типи даних
8. Методи impl — додавання поведінки до структур
9. Атрибути #[derive] — автоматична реалізація traits
10. Практика — моделювання агента БПЛА

Цілі лекції

Після цієї лекції ви зможете:

- Створювати масиви фіксованого розміру [T; N]
- Розуміти Option<T> та безпечно обробляти відсутні значення
- Використовувати match для pattern matching
- Групувати дані в кортежі та деструктуризувати їх
- Розуміти посилання & та слайси &[T]
- Створювати власні типи через struct
- Додавати методи через impl блоки
- Моделювати реальні об'єкти (агент БПЛА)

Навіщо структури даних?

У Лекції 1 ми працювали з окремими змінними:

```
let x = 10.0;  let y = 20.0;  let z = 5.0;
let battery = 85;  let is_active = true;
```

Проблема: 5 змінних для одного дрона. А якщо дронів 100?

Рішення — структури даних:

- Масиви [T; N] — багато значень одного типу
- Кортежі (T1, T2) — кілька значень різних типів
- Структури struct — іменовані поля, власні типи

ЧАСТИНА

Масиви [T; N]

Фіксована кількість елементів одного типу

Що таке масив?

Масив — послідовність елементів одного типу, що зберігаються поруч у пам'яті (контигуентно).

Ключові характеристики:

- Фіксований розмір — відомий на етапі компіляції
- Однотипні елементи — всі мають тип T
- Зберігання на стеку — швидкий доступ
- Розмір є частиною типу — [i32; 5] ≠ [i32; 6]

Синтаксис: [T; N] де T — тип, N — кількість

```
let arr: [i32; 5] = [1, 2, 3, 4, 5];
//       ^^^^^^^^ тип: масив з 5 елементів i32
```

Де живуть дані: Stack vs Heap

Stack (Стек)

- Швидке виділення пам'яті
- Розмір відомий заздалегідь
- Автоматичне звільнення
- Обмежений розмір (~1-8 МБ)

Приклади: i32, f64, bool,
масиви [T; N], кортежі

Heap (Купа)

- Повільніше виділення
- Розмір може змінюватись
- Потребує керування
- Великий розмір (ГБ)

Приклади: String, Vec<T>,
Box<T>, динамічні дані

Масиви [T; N] живуть на стеку — швидко, але розмір фіксований

Оголошення масивів

```
// Явне перелічення елементів
let days = ["Пн", "Вт", "Ср", "Чт", "Пт", "Сб", "Нд"];

// З явним типом
let numbers: [i32; 5] = [1, 2, 3, 4, 5];

// Заповнення однаковим значенням: [значення; кількість]
let zeros = [0; 10];       // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
let matrix = [[0; 3]; 3]; // матриця 3×3 нулів

// Тип виводиться автоматично
let floats = [1.0, 2.0, 3.0];  // [f64; 3]
```

[значення; кількість] — синтаксис заповнення однаковими значеннями

Доступ до елементів

```
let fruits = ["яблуко", "банан", "апельсин", "груша"];
//  індекси:      0         1          2          3

println!("{}", fruits[0]);  // яблуко
println!("{}", fruits[2]);  // апельсин

// .len() — кількість елементів (тип usize)
println!("Кількість: {}", fruits.len());  // 4

// Останній елемент
let last = fruits[fruits.len() - 1];  // груша

// Зміна (потрібен mut!)
let mut scores = [10, 20, 30];
scores[1] = 25;  // [10, 25, 30]
```

Вихід за межі масиву

Сі — невизначена поведінка

```
int arr[3] = {1, 2, 3};
int x = arr[10]; // ???
// Читає "сміття"
// Можливий segfault
// Вразливість безпеки!
```

Rust — контрольована паніка

```
let arr = [1, 2, 3];
let x = arr[10]; // ПАНІКА!

// 'index out of bounds:
//  len is 3 but index is 10'
```

Чому паніка краща?

- Програма зупиняється з чітким повідомленням
- Не працює з пошкодженими даними
- Неможливо експлуатувати як вразливість (buffer overflow)
- Легше знайти та виправити баг

ЧАСТИНА

Option<T>

Безпечна обробка відсутніх значень

Проблема NULL

У багатьох мовах (C, Java, Python) є NULL/None/null — "відсутність значення".
Проблема: компілятор НЕ змушує перевіряти на NULL!

Сі — небезпечно:

```
int* find(int arr[], int size, int target) {
    for (int i = 0; i < size; i++)
        if (arr[i] == target) return &arr[i];
    return NULL;  // не знайдено
}
int* result = find(arr, 5, 42);
printf("%d", *result);  // 💥 Якщо NULL — segfault!
```

Тоні Хоар (винахідник NULL): "Моя помилка на мільярд доларів"

Rust вирішує це через Option<T> — компілятор ЗМУШУЄ обробляти!

Що таке Option<T>?

Option<T> — enum з двома варіантами:

```
// Визначення в стандартній бібліотеці:
enum Option<T> {
    Some(T),  // Є значення типу T
    None,     // Значення відсутнє
}
```

Some(значення) — значення присутнє, "обгорнуте" в Some

None — значення відсутнє

<T> — generic (узагальнений тип):

Option<i32> — може містити i32 або бути None

Option<String> — може містити String або бути None

Ключове: Option ЗМУШУЄ явно обробити випадок відсутності!

Створення Option

```
// Some — є значення
let some_number: Option<i32> = Some(42);
let some_string: Option<&str> = Some("hello");

// None — немає значення
let no_number: Option<i32> = None;

// Тип виводиться автоматично
let x = Some(5);         // Option<i32>
let y = Some("text");    // Option<&str>

// ⚠ Для None тип ОБОВ'ЯЗКОВИЙ!
let z: Option<i32> = None;  // Який None? Для i32!

// ВАЖЛИВО: Option<i32> ≠ i32 — це РІЗНІ типи!
let a: i32 = 5;
let b: Option<i32> = Some(5);
// let c: i32 = b;  // ✕ ПОМИЛКА!
```

Що таке match?

match — конструкція для pattern matching (зіставлення з шаблоном).

Порівнює значення з набором шаблонів, виконує код першого, що підійшов.

Особливості:

- Перевіряє ВСІ можливі варіанти (exhaustive)
- Компілятор гарантує — нічого не пропущено
- Кожна гілка може повертати значення

```
let number = 3;

match number {
    1 => println!("Один"),
    2 => println!("Два"),
    3 => println!("Три"),   // ← виконається ця гілка
    _ => println!("Інше"),  // _ означає "все інше"
}
```

Синтаксис match

```
let x = 5;

// Проста форма — один вираз
match x {
    1 => println!("один"),
    2 => println!("два"),
    _ => println!("інше"),
}

// З блоками коду
match x {
    1 => {
        println!("Це одиниця!");
        println!("Спеціальне число");
    }
    2 | 3 | 4 => println!("2, 3 або 4"),  // | означає "або"
    5..=10 => println!("Від 5 до 10"),    // діапазон
    _ => println!("Щось інше"),
}

// match як вираз — повертає значення
let word = match x {
    1 => "один",
    2 => "два",
    _ => "багато",
};
```

match для Option

match ідеально підходить для Option — обробляємо обидва випадки:

```
let maybe_number: Option<i32> = Some(42);

match maybe_number {
    Some(n) => {
        // n — "розгорнуте" значення (i32), доступне тут
        println!("Знайдено: {}", n);
    }
    None => {
        println!("Значення відсутнє");
    }
}

// match як вираз
let doubled = match maybe_number {
    Some(n) => n * 2,  // є значення — подвоюємо
    None => 0,         // немає — повертаємо 0
};
println!("Результат: {}", doubled);  // 84
```

Метод .get() — безпечний доступ

.get(index) повертає Option — без паніки при виході за межі:

```
let arr = [10, 20, 30];

// Звичайний доступ — паніка!
// let x = arr[10];  // 💥 ПАНІКА!

// .get() — повертає Option, без паніки
let first = arr.get(0);   // Some(&10)
let tenth = arr.get(10);  // None — безпечно!

// Обробка через match
match arr.get(1) {
    Some(value) => println!("Знайдено: {}", value),
    None => println!("Індекс поза межами"),
}

// Або через if let
if let Some(v) = arr.get(2) {
    println!("Елемент: {}", v);
}
```

Корисні методи Option

```
let x: Option<i32> = Some(42);
let y: Option<i32> = None;

// Перевірка
x.is_some()            // true
x.is_none()            // false

// Отримання значення (ОБЕРЕЖНО!)
x.unwrap()             // 42 — але паніка якщо None!
y.unwrap()             // 💥 ПАНІКА!

// Безпечні альтернативи
x.unwrap_or(0)         // 42 (є значення)
y.unwrap_or(0)         // 0  (немає — замовчування)

x.unwrap_or_default()  // 42 (є значення)
y.unwrap_or_default()  // 0  (default для i32 = 0)

// expect — як unwrap, але з повідомленням
x.expect("Потрібне число!")  // 42
y.expect("Потрібне число!")  // 💥 паніка з повідомленням
```

if let — скорочений match

Коли цікавить тільки один варіант:

match (повний):

```
match opt {
    Some(n) => {
        println!("{}", n);
    }
    None => {}
}
```

if let (коротко):

```
if let Some(n) = opt {
    println!("{}", n);
}

// Можна додати else:
// if let Some(n) = opt {..}
// else {..}
```

if let Some(n) = opt означає: "якщо opt є Some, витягни n"

```
// Приклад з масивом
let arr = [1, 2, 3];
if let Some(first) = arr.get(0) {
    println!("Перший елемент: {}", first);
}
```

ЧАСТИНА

Кортежі (T1, T2, ...)

Групування значень різних типів

Що таке кортеж?

Кортеж (tuple) — групування фіксованої кількості значень РІЗНИХ типів.

Порівняння:

- Масив [T; N]: всі N елементів одного типу T
- Кортеж (T1, T2, T3): елементи можуть бути різних типів

```
// Масив — однотипні
let arr: [i32; 3] = [1, 2, 3];

// Кортеж — різнотипні
let tuple: (i32, f64, bool) = (42, 3.14, true);
//          ^^^  ^^^  ^^^^

// Приклад: інформація про особу
let person: (&str, i32, bool) = ("Олексій", 30, true);
//           ім'я  вік  активний
```

Створення кортежів

```
// З явним типом
let point: (f64, f64) = (10.5, 20.3);

// Тип виводиться
let rgb = (255, 128, 0);           // (i32, i32, i32)
let mixed = (1, "hello", true);    // (i32, &str, bool)

// Вкладені кортежі
let nested = ((1, 2), (3, 4));

// ⚠ Одноелементний — ПОТРІБНА КОМА!
let single = (42,);   // (i32) — кортеж
let number = (42);    // i32   — просто число!

// Unit — порожній кортеж
let unit: () = ();
```

⚠ **(42) — число, (42,) — кортеж з одного елемента!**

Доступ до елементів кортежу

Два способи: через крапку .0, .1 або деструктуризація:

```
let person = ("Олексій", 30, true);

// Спосіб 1: через КРАПКУ (не дужки!)
let name = person.0;      // "Олексій"
let age = person.1;       // 30
let active = person.2;    // true

// Спосіб 2: деструктуризація
let (name, age, active) = person;
println!("{} має {} років", name, age);

// Часткова деструктуризація — _ ігнорує
let (name, _, _) = person;  // тільки ім'я
let (_, age, _) = person;   // тільки вік
let (name, ..) = person;    // .. ігнорує решту
```

Деструктуризація детально

Деструктуризація — розпаковка структури в окремі змінні:

```
let point = (10.5, 20.3, 5.0);

// Повна
let (x, y, z) = point;

// Часткова
let (x, ..) = point;   // тільки перший
let (.., z) = point;   // тільки останній

// У циклі
let points = [(1, 2), (3, 4), (5, 6)];
for (x, y) in points {
    println!("({}, {})", x, y);
}

// У параметрах функції
fn distance((x1, y1): (f64, f64), (x2, y2): (f64, f64)) -> f64 {
    ((x2-x1).powi(2) + (y2-y1).powi(2)).sqrt()
}
```

Повернення кількох значень

Кортеж дозволяє повертати кілька значень з функції:

```
// Функція повертає мінімум і максимум
fn min_max(numbers: &[i32]) -> (i32, i32) {
    let mut min = numbers[0];
    let mut max = numbers[0];
    for &n in numbers {
        if n < min { min = n; }
        if n > max { max = n; }
    }
    (min, max)  // повертаємо кортеж
}

fn main() {
    let arr = [3, 1, 4, 1, 5, 9, 2, 6];
    let (min, max) = min_max(&arr);  // деструктуризація
    println!("Мін: {}, Макс: {}", min, max);
}
```

ЧАСТИНА

Посилання &T

Доступ до даних без передачі володіння

Що таке посилання?

Посилання (reference) — адреса в пам'яті, де знаходяться дані.

На відміну від вказівників у Сі, посилання в Rust завжди валідні.

& — створює посилання (borrowing, запозичення)

* — розіменування (доступ до даних)

```
let x = 5;
let ref_x = &x;     // ref_x — посилання на x

println!("x = {}", x);           // 5
println!("ref_x = {}", ref_x);   // 5 (авто-розіменування)
println!("*ref_x = {}", *ref_x); // 5 (явне розіменування)

// Посилання НЕ копіює дані!
// ref_x "дивиться" на ту саму пам'ять що й x
```

Навіщо потрібні посилання?

Передача великих даних без копіювання:

```
// БЕЗ посилання — копіювання (повільно для великих даних!)
fn print_copy(arr: [i32; 1000]) {
    println!("{:?}", arr);
}

// З посиланням — передаємо тільки адресу (швидко!)
fn print_ref(arr: &[i32; 1000]) {
    println!("{:?}", arr);
}

fn main() {
    let big = [0; 1000];

    // print_copy(big);  // копіює 4000 байт!
    print_ref(&big);     // передає 8 байт (адреса)
}
```

& перед змінною — створює посилання, & в типі — очікує посилання

&T vs &mut T

&T — тільки читання

```
let x = 5;
let r = &x;
println!("{}", r); // OK
// *r = 10; // ✕ не можна!

// Можна багато &T
let r1 = &x;
let r2 = &x;  // OK
```

&mut T — читання і запис

```
let mut x = 5;
let r = &mut x;
*r = 10;  // OK
println!("{}", x); // 10

// Тільки ОДНЕ &mut T!
// let r2 = &mut x; // ✕
```

Правило: багато &T АБО один &mut T — ніколи разом!

ЧАСТИНА

Слайси &[T]

"Вікно" в послідовність даних

Що таке слайс?

Слайс (slice) — посилання на неперервну частину послідовності.

Характеристики:

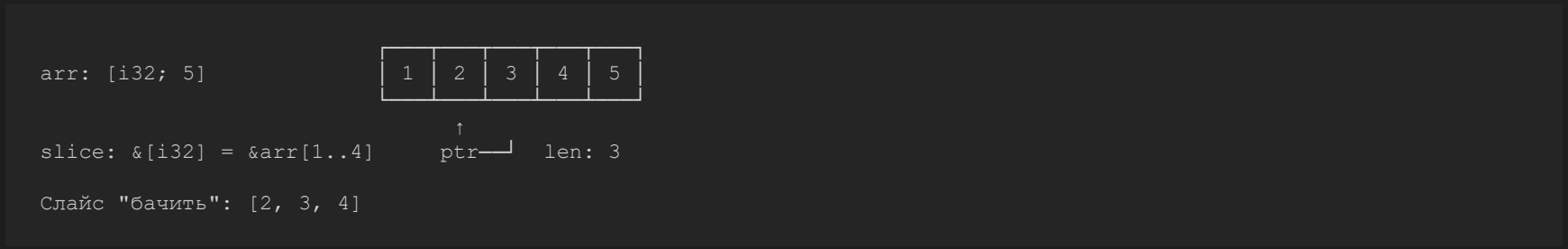
- НЕ володіє даними — тільки "дивиться"
- Розмір НЕ частина типу (на відміну від масиву)
- &[T] може посилатись на частину [T; N] будь-якого N

[i32; 5] — масив, розмір відомий компілятору

```
let arr: [i32; 5] = [1, 2, 3, 4, 5];
let slice: &[i32] = &arr[1..4];  // [2, 3, 4]
//         ^^^^^ тип НЕ містить розміру
```

Fat Pointer — як влаштований слайс

Слайс — "fat pointer" (товстий вказівник), містить ДВА значення:
1. Вказівник на перший елемент
2. Довжина (кількість елементів)



Розмір &[T]: 16 байт (8 ptr + 8 len) на 64-біт системі

Завдяки len слайс знає свою довжину — .len() працює!

Синтаксис діапазонів

```
let arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
//         0  1  2  3  4  5  6  7  8  9

&arr[2..5]    // [2, 3, 4]       — від 2 до 5 (НЕ включно)
&arr[2..=5]   // [2, 3, 4, 5]    — від 2 до 5 (включно)
&arr[..5]     // [0, 1, 2, 3, 4] — від початку
&arr[5..]     // [5, 6, 7, 8, 9] — до кінця
&arr[..]      // весь масив

// Приклади
let first = &arr[..3];   // [0, 1, 2]
let last = &arr[7..];    // [7, 8, 9]
let mid = &arr[3..7];    // [3, 4, 5, 6]
```

start..end — НЕ включає end, start..=end — включає end

Слайси як параметри функцій

Функція з &[T] працює з масивами будь-якого розміру:

```
fn sum(numbers: &[i32]) -> i32 {
    let mut total = 0;
    for &n in numbers {
        total += n;
    }
    total
}

fn main() {
    let arr3 = [1, 2, 3];
    let arr5 = [10, 20, 30, 40, 50];

    println!("{}", sum(&arr3));       // 6
    println!("{}", sum(&arr5));       // 150
    println!("{}", sum(&arr5[1..4])); // 90 (20+30+40)
}
```

Мутабельні слайси &mut [T]

```
fn double_all(numbers: &mut [i32]) {
    for n in numbers {
        *n *= 2;  // n: &mut i32, *n — розіменування
    }
}

fn main() {
    let mut arr = [1, 2, 3, 4, 5];

    println!("До:    {:?}", arr);  // [1, 2, 3, 4, 5]

    double_all(&mut arr);

    println!("Після: {:?}", arr);  // [2, 4, 6, 8, 10]

    // Частина масиву
    let mut arr2 = [1, 2, 3, 4, 5];
    double_all(&mut arr2[1..4]);
    println!("{:?}", arr2);  // [1, 4, 6, 8, 5]
}
```

Рядкові слайси &str

&str — слайс байтів UTF-8 тексту:

```
let s = String::from("Hello, Rust!");

let hello: &str = &s[0..5];   // "Hello"
let rust: &str = &s[7..11];   // "Rust"

// Рядковий літерал — це теж &str
let greeting: &str = "Hello";
```

⚠ **УВАГА: індекси — це БАЙТИ, не символи!**

```
Кирилиця займає 2 байти на символ:
"Привіт"[0..1] — 💥 ПАНІКА! (розріже 'П' навпіл)
"Привіт"[0..2] — OK, поверне "П"
"Привіт"[0..12] — OK, весь рядок (6 символів × 2 байти)
```

ЧАСТИНА

Структури struct

Власні типи даних з іменованими полями

Що таке структура?

Структура (struct) — користувацький тип з іменованими полями.

Переваги над кортежами:

- Поля мають імена — код самодокументується
- Можна додавати методи через impl
- Можна реалізовувати traits

```
// Кортеж — незрозуміло
let drone: (f64, f64, f64, u8, bool) = (0.0, 0.0, 0.0, 100, true);
// Що таке drone.3?

// Структура — зрозуміло!
struct Drone {
    x: f64, y: f64, z: f64,
    battery: u8,
    is_active: bool,
}
```

Оголошення структури

```
// Структура оголошується ПОЗА функціями
struct Point {
    x: f64,
    y: f64,
}

struct Person {
    name: String,
    age: u8,
    is_student: bool,
}

struct Rectangle {
    width: u32,
    height: u32,
}
```

Конвенція: назва — PascalCase, поля — snake_case

Створення екземпляра

```
struct Point { x: f64, y: f64 }

fn main() {
    // ВСІ поля обов'язкові!
    let origin = Point { x: 0.0, y: 0.0 };

    // Порядок не важливий
    let p = Point { y: 20.5, x: 10.0 };

    // Доступ через крапку
    println!("x={}, y={}", p.x, p.y);

    // Зміна (потрібен mut!)
    let mut p2 = Point { x: 1.0, y: 2.0 };
    p2.x = 5.0;  // OK
}
```

Скорочений синтаксис

```
struct Point { x: f64, y: f64 }

// Якщо ім'я змінної = ім'я поля
fn create(x: f64, y: f64) -> Point {
    Point { x, y }  // замість Point { x: x, y: y }
}

// Копіювання полів з іншої структури
fn main() {
    let p1 = Point { x: 1.0, y: 2.0 };

    // Нова структура, але змінюємо тільки x
    let p2 = Point {
        x: 5.0,
        ..p1  // решта з p1
    };

    println!("({}, {})", p2.x, p2.y);  // (5, 2)
}
```

Атрибут #[derive()]

#[derive(...)] — автоматично реалізує стандартні traits:

```
// Без derive
struct Point { x: f64, y: f64 }
let p = Point { x: 1.0, y: 2.0 };
// println!("{:?}", p);  // ✕ ПОМИЛКА!

// З derive(Debug)
#[derive(Debug)]
struct Point2 { x: f64, y: f64 }
let p2 = Point2 { x: 1.0, y: 2.0 };
println!("{:?}", p2);   // Point2 { x: 1.0, y: 2.0 }
println!("{:#?}", p2);  // красивий формат

// Можна комбінувати
#[derive(Debug, Clone, Copy, PartialEq)]
struct Point3 { x: f64, y: f64 }
```

Методи: impl блок

```
struct Rectangle { width: u32, height: u32 }

impl Rectangle {
    // Метод — перший параметр &self
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn perimeter(&self) -> u32 {
        2 * (self.width + self.height)
    }
}

fn main() {
    let rect = Rectangle { width: 10, height: 5 };
    println!("Площа: {}", rect.area());       // 50
    println!("Периметр: {}", rect.perimeter()); // 30
}
```

Типи self

```
impl Rectangle {
    // &self — тільки читання
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // &mut self — може змінювати
    fn double(&mut self) {
        self.width *= 2;
        self.height *= 2;
    }

    // self — забирає ownership
    fn into_square(self) -> Rectangle {
        let side = (self.width + self.height) / 2;
        Rectangle { width: side, height: side }
    }
}
```

&self — читає, &mut self — змінює, self — забирає

Асоційовані функції (конструктори)

Функції в impl БЕЗ self — викликаються через Type::func():

```
impl Rectangle {
    // Конструктор — немає self!
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle { width, height }
    }

    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}

fn main() {
    // Виклик через :: (не крапку!)
    let rect = Rectangle::new(10, 5);
    let sq = Rectangle::square(7);

    println!("{}", rect.area());  // 50
    println!("{}", sq.area());    // 49
}
```

Tuple structs

Структури без іменованих полів:

```
struct Color(u8, u8, u8);         // RGB
struct Point3D(f64, f64, f64);    // x, y, z

fn main() {
    let red = Color(255, 0, 0);
    let origin = Point3D(0.0, 0.0, 0.0);

    // Доступ через індекс
    println!("R={}", red.0);

    // Деструктуризація
    let Color(r, g, b) = red;
}

// Newtype pattern — різні типи з однаковим вмістом
struct Meters(f64);
struct Seconds(f64);
// Meters і Seconds — РІЗНІ типи!
```

ЧАСТИНА

Практика

Моделювання агента БПЛА

Моделювання дрона: структури

```
#[derive(Debug)]
struct Position {
    x: f64,
    y: f64,
    z: f64,
}

#[derive(Debug)]
struct Drone {
    id: u32,
    position: Position,  // вкладена структура
    battery: u8,
    is_active: bool,
}

impl Position {
    fn new(x: f64, y: f64, z: f64) -> Position {
        Position { x, y, z }
    }
    fn origin() -> Position {
        Position { x: 0.0, y: 0.0, z: 0.0 }
    }
}
```

Моделювання дрона: методи

```
impl Drone {
    fn new(id: u32) -> Drone {
        Drone {
            id,
            position: Position::origin(),
            battery: 100,
            is_active: false,
        }
    }

    fn activate(&mut self) {
        if self.battery > 10 {
            self.is_active = true;
            println!("Дрон {} активовано", self.id);
        } else {
            println!("Недостатньо заряду!");
        }
    }

    fn status(&self) {
        println!("Дрон {}: {}%, active={}",
            self.id, self.battery, self.is_active);
    }
}
```

Моделювання дрона: переміщення

```
impl Drone {
    fn move_to(&mut self, x: f64, y: f64, z: f64) {
        if !self.is_active {
            println!("Дрон не активний!");
            return;
        }

        let dx = x - self.position.x;
        let dy = y - self.position.y;
        let dz = z - self.position.z;
        let dist = (dx*dx + dy*dy + dz*dz).sqrt();

        let drain = (dist / 10.0) as u8;  // 1% на 10м
        if drain > self.battery {
            println!("Недостатньо заряду!");
            return;
        }

        self.position = Position::new(x, y, z);
        self.battery -= drain;
        println!("Дрон {} → ({},{},{})", self.id, x, y, z);
    }
}
```

Повний приклад

```
fn main() {
    let mut drone = Drone::new(1);

    drone.status();       // Дрон 1: 100%, active=false
    drone.activate();     // Дрон 1 активовано

    drone.move_to(100.0, 50.0, 30.0);
    drone.move_to(150.0, 80.0, 25.0);

    drone.status();       // Дрон 1: 85%, active=true

    println!("{:#?}", drone);
}
// Drone {
//     id: 1,
//     position: Position { x: 150, y: 80, z: 25 },
//     battery: 85,
//     is_active: true,
// }
```

Завдання для самостійної роботи

1. Масив та Option

   Напишіть fn find(arr: &[i32], target: i32) -> Option<usize>

   що повертає індекс елемента або None.

2. Структура Student

   Поля: name, age, grades: [u8; 5].

   Методи: new(), average() -> f64, is_excellent() -> bool.

3. Rectangle з contains

   Методи: area(), perimeter(), contains(point: &Point) -> bool.

4. Рій дронів

   fn find_nearest(drones: &[Drone], x, y, z) -> Option<usize>

   Повертає індекс найближчого дрона.

Підсумок

Масиви [T; N]	Stack, фіксований
Option<T>	Some/None
match	Pattern matching
Кортежі	Різні типи
Посилання &T	Borrowing
Слайси &[T]	Fat pointer
struct	Власні типи
impl	Методи

Наступна лекція:

Ownership — унікальна система володіння пам'яттю Rust.
Чому String переміщується, а i32 копіюється? Move vs Copy.