Ownership

Система володіння — серце Rust

Move • Copy • Clone • Drop • Borrowing

Лекція 3 (розширена версія)

Бичков Олексій Сергійович

Чому Ownership — найважливіша тема?

Ownership — це те, що робить Rust унікальним серед мов програмування.

Це система керування пам'яттю, яка:
- Не потребує збирача сміття (Garbage Collector)
- Гарантує відсутність витоків пам'яті
- Запобігає data races на етапі компіляції
- Забезпечує memory safety без runtime overhead

Якщо ви зрозумієте Ownership — ви зрозумієте Rust.
Якщо ні — будете постійно "боротися" з компілятором.

⚠ Ця лекція потребує уважного вивчення.
Перегляньте її кілька разів, поки концепції не стануть інтуїтивними.

Зміст лекції

Частина 1: Проблема керування пам'яттю
- Як інші мови вирішують цю проблему
- Чому Rust обрав інший шлях

Частина 2: Три правила Ownership
- Кожне значення має власника
- Тільки один власник одночасно
- Значення звільняється при виході власника зі scope

Частина 3: Move — передача володіння
- Що таке move semantics
- String vs &str — глибоке пояснення

Частина 4: Copy та Clone

Частина 5: Borrowing — запозичення

Частина 6: Практика з помилками компілятора

Цілі лекції

Після цієї лекції ви зможете:

- Пояснити три правила ownership своїми словами
- Розуміти різницю між move та copy
- Знати, які типи копіюються, а які переміщуються
- Використовувати Clone для явного копіювання
- Розуміти borrowing (&T та &mut T)
- Читати та виправляти помилки borrow checker
- Писати код, що "дружить" з ownership системою

ЧАСТИНА 1

Проблема керування пам'яттю

Як різні мови вирішують цю проблему

Що таке керування пам'яттю?

Програма використовує пам'ять для зберігання даних.
Коли дані більше не потрібні — пам'ять треба звільнити.

Ключові питання:
- Коли виділяти пам'ять?
- Коли звільняти пам'ять?
- Хто відповідальний за звільнення?

Проблеми при неправильному керуванні:

Memory Leak (витік)

Пам'ять виділена, але ніколи не звільняється. Програма "з'їдає" всю пам'ять.

Use After Free

Пам'ять звільнена, але програма намагається її використати.

Double Free — звільнення тієї самої пам'яті двічі → невизначена поведінка

Реальні наслідки помилок пам'яті

Статистика вразливостей (офіційні дані):

- Microsoft (2019): ~70% вразливостей Windows — помилки пам'яті
- Google (2020): ~70% критичних багів Chrome — помилки пам'яті
- Android: більшість критичних CVE пов'язані з memory safety

Відомі атаки:
- Heartbleed (2014) — buffer over-read в OpenSSL
- WannaCry (2017) — експлуатація use-after-free
- Stagefright (2015) — RCE через медіа-файли

Чому важко уникнути в C/C++:
- Компілятор НЕ перевіряє правильність роботи з пам'яттю
- Помилки виявляються в runtime (якщо пощастить)
- Код може працювати роками і раптом зламатись

Rust створений, щоб зробити ці помилки НЕМОЖЛИВИМИ!

Підхід C: Ручне керування

```c
#include <stdlib.h>

int main() {
    char* message = malloc(100);  // Виділяємо вручну
    strcpy(message, "Hello!");
    printf("%s\n", message);

    free(message);  // ОБОВ'ЯЗКОВО звільнити!

    // ✕ Небезпека: message все ще вказує на звільнену пам'ять
    // printf("%s\n", message);  // Use After Free!

    // ✕ Небезпека: подвійне звільнення
    // free(message);  // Double Free!
    return 0;
}
```

C: повний контроль, але вся відповідальність на програмісті.
Одна помилка — і вразливість безпеки або крах програми.

Типові помилки в C — детальніше

```
// Use After Free — класична помилка
char* get_message() {
    char buffer[100];           // Локальний масив
    strcpy(buffer, "Hello");
    return buffer;              // 💥 Повертаємо вказівник на локальну змінну!
}                               // buffer знищується!

// Double Free — ще одна класика
void process(char* data) {
    // ... робота з data ...
    free(data);                 // Звільнили тут
}

int main() {
    char* msg = malloc(100);
    process(msg);               // process звільнив msg
    free(msg);                  // 💥 Double Free!
}
```

В C компілятор НЕ попереджає — помилки виявляються тільки в runtime!

Підхід Java/Python/Go: Garbage Collector

```
// Java — GC автоматично звільняє пам'ять
public class Main {
    public static void main(String[] args) {
        String message = new String("Hello!");
        System.out.println(message);
        // Не потрібно нічого звільняти!
        // GC сам очистить, коли message не використовується
    }
}

# Python — теж GC
message = "Hello!"
print(message)
# Автоматичне очищення
```

Переваги GC:

- Простота
- Безпека — немає use after free

Недоліки GC:

- Паузи — GC зупиняє програму
- Накладні витрати — більше пам'яті

Проблеми Garbage Collection — детально

1. Stop-The-World паузи
   GC періодично зупиняє ВСЮ програму для аналізу пам'яті
   • Java: типові паузи 10-100мс, інколи секунди
   • Критично для: ігор (60 FPS = 16мс/кадр), торгових систем

2. Непередбачуваність
   Ви не знаєте, КОЛИ GC запуститься
   Програма може "замерзнути" в найгірший момент

3. Більше використання пам'яті
   GC працює ефективніше, коли пам'яті багато
   Типово програми з GC використовують 2-3x більше RAM

4. Складність налаштування
   Сучасні GC мають десятки параметрів для тюнінгу
   Неправильні налаштування = ще гірша продуктивність

Підхід Rust: Ownership

Rust обрав третій шлях: Ownership System

Ідея: компілятор СТАТИЧНО відстежує, хто "володіє" кожним значенням,
і автоматично вставляє код звільнення пам'яті.

```
fn main() {
    let message = String::from("Hello!");
    println!("{}", message);

    // Тут message виходить зі scope
    // Rust АВТОМАТИЧНО звільняє пам'ять
    // Ніякого GC, ніякого ручного free!
}
```

✓ Безпека як у GC + ✓ Швидкість як у C + ✓ Нуль накладних витрат

Порівняння трьох підходів

Підхід	Безпека	Швидкість	Простота
C (ручне)	✗ Низька	✓ Максимальна	✗ Складно
Java/Python (GC)	✓ Висока	✗ Паузи GC	✓ Просто
Rust (Ownership)	✓ Висока	✓ Максимальна	⚠ Треба вчити

Rust вимагає зусиль на початку, але дає найкращий результат:

- Компілятор перевіряє все за вас
- Якщо код скомпілювався — він безпечний

ЧАСТИНА 2

Три правила Ownership

Фундамент системи володіння

Правило 1: Кожне значення має власника

Кожне значення в Rust має змінну — його власника (owner).

```
let s = String::from("hello");
//  ^   ^^^^^^^^^^^^^^^^^^^^^
//  |   значення (дані на heap)
//  |
//  власник (змінна s)

let x = 42;
//  ^   ^^
//  |   значення
//  власник
```

Метафора: ресурс — це фізичний об'єкт (ключі).
Ключі завжди хтось тримає — вони не висять у повітрі.

Метафора: Власник квартири

Уявіть, що значення — це квартира, а змінна — власник:

• Квартира завжди має ОДНОГО офіційного власника
(навіть якщо там живуть орендарі)

• Власник ВІДПОВІДАЄ за квартиру:
- Платить податки, комунальні
- Робить ремонт
- Вирішує, коли продати

• Коли власник "йде" — квартира переходить новому власнику
або звільняється (знищується)

• Квартира НЕ МОЖЕ бути "нічийною" — завжди є власник

Так само в Rust: дані завжди мають власника (змінну).

Правило 2: Тільки один власник

У кожен момент часу може бути тільки ОДИН власник значення.

```
let s1 = String::from("hello");
let s2 = s1;  // Володіння ПЕРЕХОДИТЬ від s1 до s2

// s1 більше НЕ є власником!
// println!("{}", s1);  // ✕ ПОМИЛКА компіляції!
println!("{}", s2);     // ✓ OK — s2 тепер власник
```

Метафора: передаєте ключі іншій людині.
Тепер ви не можете відкрити двері — ключі не у вас.

Це називається MOVE — значення "переїхало" до іншого власника.

Чому тільки ОДИН власник?

Уявіть, що було б з кількома власниками:

Проблема 1: Хто звільняє пам'ять?
- Власник A: "Я звільню"
- Власник B: "Ні, я звільню"
- Результат: Double Free або Memory Leak

Проблема 2: Use After Free
- Власник A звільнив пам'ять
- Власник B не знає і намагається використати
- Результат: крах програми

Проблема 3: Data Race (багатопотоковість)
- Власник A змінює дані
- Власник B читає ті самі дані одночасно
- Результат: непередбачуваний стан

Один власник = чітка відповідальність!

Правило 3: Drop при виході зі scope

Коли власник виходить зі scope, значення автоматично звільняється.

```
fn main() {
    {   // Починається новий scope
        let s = String::from("hello");
        println!("{}", s);
    }   // s виходить зі scope → drop
        // Пам'ять АВТОМАТИЧНО звільнена!

    // s тут не існує
    // println!("{}", s);  // ✕ ПОМИЛКА!
}
```

Компілятор автоматично вставляє drop(s) перед }

Що таке Scope (область видимості)?

Scope — це регіон коду, де змінна є дійсною (valid).

```
fn main() {                    // main scope
    let a = 1;                 // a дійсна

    {                          // внутрішній scope
        let b = 2;             // b дійсна тут
        println!("{}", a + b);
    }                          // b drop

    // println!("{}", b);      // ✕ b не існує!
    println!("{}", a);         // ✓ a ще дійсна
}                              // a drop
```

Scope визначається { }. Функція, if, loop, match — створюють scope.

Drop — як це працює всередині

```
// Drop — це trait, який можна реалізувати
struct Resource { name: String }

impl Drop for Resource {
    fn drop(&mut self) {
        println!("Звільняю: {}", self.name);
    }
}

fn main() {
    let r1 = Resource { name: String::from("Файл") };
    let r2 = Resource { name: String::from("З'єднання") };
    println!("Використовую...");
}   // drop(r2), потім drop(r1) — LIFO!

// Вивід:
// Використовую...
// Звільняю: З'єднання
// Звільняю: Файл
```

LIFO: останній створений — перший знищується

Три правила Ownership — підсумок

1. **Кожне значення має власника**
2. **Тільки один власник у кожен момент**
3. **Вихід зі scope → значення звільняється**

Ці три правила — все, що потрібно знати про Ownership.
Решта — це наслідки цих правил.

Компілятор Rust перевіряє дотримання цих правил
на етапі компіляції — тому немає runtime помилок!

ЧАСТИНА 3

Move — передача володіння

Що відбувається при присвоєнні

Нагадування: Stack vs Heap

Stack (Стек)

- Фіксований розмір
- Швидкий доступ (LIFO)
- Автоматичне керування

Типи: i32, f64, bool, char,
[T; N], кортежі

Heap (Купа)

- Динамічний розмір
- Повільніший доступ
- Потребує керування

Типи: String, Vec<T>,
Box<T>, HashMap

Ownership важливий для heap даних — там реальне виділення/звільнення

String — як влаштований

```
let s = String::from("hello");
```

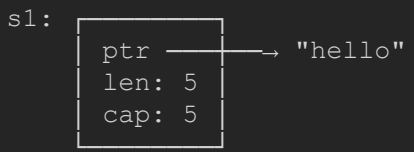
(24 байти на stack)

ptr — вказівник на дані в heap
len — поточна довжина рядка
capacity — скільки виділено пам'яті

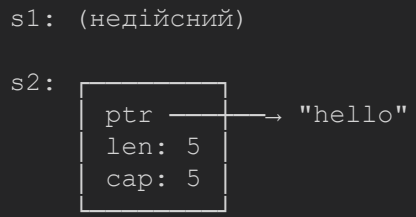
Що відбувається при move?

```
let s1 = String::from("hello");
let s2 = s1;  // MOVE!
```

До move:



Після move:



Копіюються тільки 24 байти на stack (ptr, len, cap).
Дані на heap НЕ копіюються.
s1 стає недійсним — компілятор забороняє його використання.

Чому Rust не копіює heap дані?

Проблема 1: ПОВІЛЬНО
Копіювання 1GB при кожному присвоєнні = катастрофа

Проблема 2: Double Free
Якщо s1 і s2 вказують на ту саму пам'ять:
- s1 виходить зі scope → free()
- s2 виходить зі scope → free() ← 💥

Проблема 3: Data Race
Два власники можуть одночасно змінювати дані

Рішення: при присвоєнні — MOVE, не copy.
Старий власник "забуває" про дані → проблеми зникають!

Move при передачі в функцію

```
fn take_ownership(s: String) {
    println!("{}", s);
}   // s drop

fn main() {
    let s1 = String::from("hello");

    take_ownership(s1);  // s1 переміщується

    // println!("{}", s1);  // ✕ ПОМИЛКА!
    // s1 вже не дійсний
}
```

Функція "забирає" значення у власність.

Move при поверненні з функції

```
fn create_string() -> String {
    let s = String::from("hello");
    s  // Володіння передається назовні
}

fn take_and_give_back(s: String) -> String {
    println!("Отримав: {}", s);
    s  // Повертаємо назад
}

fn main() {
    let s1 = create_string();
    let s2 = take_and_give_back(s1);
    println!("{}", s2);  // ✓ OK
}
```

String vs &str — ключова різниця

String

- Володіє даними
- На heap
- Може змінюватись
- При присвоєнні — MOVE

```
let s = String::from("hi");
let s2 = s;  // move!
```

&str (slice)

- НЕ володіє
- "Дивиться" на дані
- Read-only
- При присвоєнні — COPY

```
let s: &str = "hi";
let s2 = s;  // copy!
```

"hello" (літерал) — це &str в бінарнику

ЧАСТИНА 4

Copy та Clone

Типи, що копіюються vs типи, що переміщуються

Copy trait — автоматичне копіювання

```
let x = 5;
let y = x;  // COPY! (не move)

println!("x = {}", x);  // ✓ OK!
println!("y = {}", y);  // ✓ OK!

// Обидві змінні дійсні
```

Copy типи — типи, які:
- Живуть повністю на stack
- Копіювання — просте побітове копіювання
- Копіювання швидке і дешеве

Copy — це trait, який позначає такі типи

Які типи мають Copy?

Copy типи (копіюються):

- i8..i128, u8..u128
- f32, f64
- bool
- char
- (i32, f64) — кортежі
- [i32; 10] — масиви
- &T — посилання

НЕ Copy типи (move):

- String
- Vec<T>
- HashMap
- Box<T>
- File, TcpStream
- Структури з не-Copy

Правило: якщо тип має heap-дані — він НЕ Copy

Clone trait — явне копіювання

```
let s1 = String::from("hello");
let s2 = s1.clone();  // ЯВНЕ глибоке копіювання

println!("s1 = {}", s1);  // ✓ OK!
println!("s2 = {}", s2);  // ✓ OK!

// Дві незалежні копії на heap
```

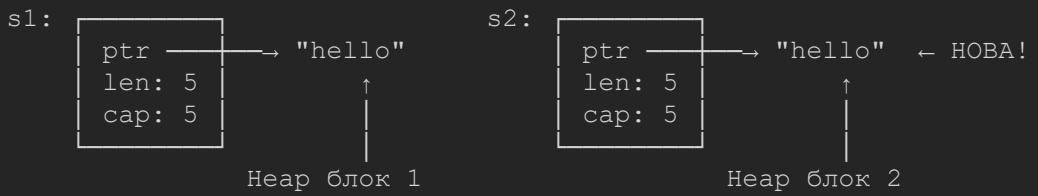
Clone vs Copy:
• Copy — автоматичне, неявне, дешеве
• Clone — явне (.clone()), може бути дороге

Типи з Copy автоматично мають Clone.
Типи з Clone не обов'язково мають Copy.

Clone — що відбувається в пам'яті

```
let s1 = String::from("hello");
let s2 = s1.clone();
```

Після clone():



Дві НЕЗАЛЕЖНІ копії даних.

.clone() виділяє нову пам'ять — може бути повільно!

Додавання Copy/Clone до власних типів

```
#[derive(Copy, Clone, Debug)]
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    let p1 = Point { x: 1.0, y: 2.0 };
    let p2 = p1;  // COPY!

    println!("p1: {:?}", p1);  // ✓ OK
    println!("p2: {:?}", p2);  // ✓ OK
}

// ✕ Не можна Copy якщо є не-Copy поля:
// struct Person { name: String }  // String не Copy!
```

ЧАСТИНА 5

Borrowing — запозичення

Доступ до даних без передачі володіння

Проблема: як передати без втрати?

```
fn calculate_length(s: String) -> usize {
    s.len()
}   // s drop!

fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(s1);  // s1 moved

    // println!("{}", s1);  // ✕ ПОМИЛКА!
}
```

Move "забирає" значення. Але що якщо воно ще потрібне?

Рішення: Borrowing (запозичення)

```
fn calculate_length(s: &String) -> usize {  // &String
    s.len()
}   // s (посилання) drop, дані НЕ звільняються

fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);  // &s1 — позичаємо

    println!("'{}' має {} символів", s1, len);  // ✓ OK!
}
```

& — створює посилання (reference)
&s1 — "позичити s1 для читання"

Іммутабельні посилання &T

```
let s = String::from("hello");

let r1 = &s;  // OK
let r2 = &s;  // OK — багато &T одночасно!
let r3 = &s;  // OK

println!("{}", r1);  // ✓
println!("{}", r2);  // ✓

// r1.push_str("!");  // ✗ Не можна змінювати!
```

&T: скільки завгодно одночасно, тільки читання

Мутабельні посилання &mut T

```
fn add_world(s: &mut String) {
    s.push_str(" world");  // ✓ Можемо змінювати!
}

fn main() {
    let mut s = String::from("hello");  // mut!

    add_world(&mut s);

    println!("{}", s);  // "hello world"
}
```

Для &mut T потрібно:
• let mut s • &mut s • fn f(s: &mut String)

Головне правило Borrowing

В один момент часу можна мати АБО:
- **Скільки завгодно &T**
- **ОДНЕ &mut T**

Ніколи обидва одночасно!

```
let mut s = String::from("hello");

let r1 = &s;      // OK
let r2 = &s;      // OK
// let r3 = &mut s;  // ✕ Не можна поки є &T!

println!("{}", r1);
// r1, r2 більше не використовуються

let r3 = &mut s;  // ✓ OK тепер
```

Чому таке обмеження?

Це правило запобігає DATA RACE на етапі компіляції:

Data Race виникає, коли:
1. Кілька вказівників доступаються до тих самих даних
2. Хоча б один пише
3. Немає синхронізації

Правило Rust робить data race НЕМОЖЛИВИМ:
- Багато &T — ОК (ніхто не змінює)
- Один &mut T — ОК (ніхто не читає)
- &T + &mut T — ЗАБОРОНЕНО!

В інших мовах такі баги шукають місяцями. В Rust — компілятор за секунди!

Приклад конфлікту borrowing

```
let mut v = vec![1, 2, 3];

let first = &v[0];  // &T

v.push(4);  // ✕ push потребує &mut v!

println!("{}", first);
```

```
error[E0502]: cannot borrow `v` as mutable because it is
              also borrowed as immutable
```

Рішення: використати first ДО push, або обмежити scope.

Dangling References — висячі посилання

```
fn dangle() -> &String {     // ✕ ПОМИЛКА!
    let s = String::from("hello");
    &s  // Повертаємо посилання
}       // s drop! Посилання → нікуди

// Правильно:
fn no_dangle() -> String {   // ✓ OK
    let s = String::from("hello");
    s   // Передаємо володіння
}
```

Rust не дозволяє посиланню пережити дані, на які воно вказує.

ЧАСТИНА 6

Практика з помилками

Вчимося читати повідомлення компілятора

Помилка 1: Use after move

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}", s1);  // ✕
```

```
error[E0382]: borrow of moved value: `s1`
  value moved here... value borrowed here after move
```

Рішення:

```
// Варіант 1: Clone
let s2 = s1.clone();

// Варіант 2: Використати s2
println!("{}", s2);
```

Помилка 2: Cannot borrow as mutable

```
let s = String::from("hello");
s.push_str(" world");  // ✕
```

```
error[E0596]: cannot borrow `s` as mutable
```

Рішення:

```
let mut s = String::from("hello");  // Додати mut!
s.push_str(" world");  // ✓ OK
```

Помилка 3: Mutable + immutable borrow

```
let mut s = String::from("hello");
let r1 = &s;
let r2 = &mut s;  // ✕
println!("{}", r1);
```

```
error[E0502]: cannot borrow `s` as mutable
```

Рішення: використати r1 до створення r2:

```
println!("{}", r1);  // Використали
let r2 = &mut s;     // ✓ OK тепер
```

Практика: передача агента

```
struct Drone { id: u32, battery: u8 }

// ✕ Забирає володіння:
fn report(drone: Drone) { println!("{}", drone.id); }

fn main() {
    let d = Drone { id: 1, battery: 85 };
    report(d);
    // report(d);  // ✕ moved!
}

// ✓ Рішення: посилання
fn report(drone: &Drone) { ... }
report(&d);  // ✓
report(&d);  // ✓ Можна знову!
```

Типові паттерни роботи з Ownership

1. Тільки читання — &T:
   fn print_info(drone: &Drone) { ... }

2. Потрібно змінити — &mut T:
   fn charge(drone: &mut Drone) { drone.battery = 100; }

3. Новий власник — T:
   fn destroy(drone: Drone) { ... }

4. Незалежна копія — .clone():
   let backup = drone.clone();

За замовчуванням: &T. Змінювати: &mut T. T — рідко.

Завдання для самостійної роботи

1. Move semantics
   Напишіть код з move для String.
   Виправте помилку: clone та посилання.

2. Copy types
   Створіть Point { x: f64, y: f64 } з derive(Copy, Clone).

3. Borrowing
   Напишіть функції для Drone:
   - get_battery(&self) -> u8
   - set_battery(&mut self, value: u8)
   - destroy(self)

4. Виправлення помилок
   Знайдіть та виправте 3 помилки ownership.

Підсумок: Ownership Rules

1. Кожне значення має власника

2. Тільки один власник одночасно

3. Вихід зі scope → drop

Move vs Copy:
• Copy (i32, f64, bool) — копіюються автоматично
• Не-Copy (String, Vec) — move
• .clone() — явне копіювання

Borrowing: &T (багато, читання) | &mut T (один, запис)

Підсумок: Borrowing Rules

АБО будь-яка кількість &T
АБО рівно одне &mut T
Ніколи обидва!

Це запобігає:
- Data races
- Iterator invalidation
- Use after free
- Dangling pointers

Компілятор — ваш друг! Він знаходить помилки ДО production.

Ownership vs інші підходи — фінал

C/C++:
malloc/free → 70% вразливостей

Java/Python/Go:
GC → паузи, overhead, непередбачуваність

Rust:
Ownership + borrow checker → безпека БЕЗ overhead
КОМПІЛЯТОР перевіряє!
Якщо скомпілювалось — воно безпечне!

Вартість Rust:
- Крива навчання
- Іноді "боротьба" з borrow checker
- АЛЕ: ви вчитеся писати КРАЩИЙ код у будь-якій мові!

Наступна лекція

Enum та Pattern Matching

- Enum — більше ніж перелік
- Варіанти з даними
- Потужність match
- Option<T> та Result<T, E>
- Машина станів агента БПЛА

Enum в Rust — це Algebraic Data Types.
Ідеальний інструмент для моделювання станів!

 Дякую за увагу!