Лекція 11

Lifetimes: Час життя посилань

Гарантії безпеки пам'яті на етапі компіляції

'a • 'static • elision • bounds • annotations

🤖 Приклади: посилання на стан агента, view патерни

Частина 1: Основи lifetimes

О. С. Бичков • 2025

План лекції (Частина 1)

1. Проблема dangling references
2. Що таке lifetime?
3. Borrow checker та lifetimes
4. Lifetime annotations синтаксис
5. Lifetimes у функціях
6. Чому потрібні анотації?
7. Множинні lifetimes
8. Lifetime elision rules
9. Правило 1: input lifetimes
10. Правило 2: single input
11. Правило 3: &self
12. Коли elision не працює
13. 🤖 Функції з посиланнями на агентів
14. 🤖 Порівняння позицій
15. Візуалізація lifetimes

Частина 2: Lifetimes у структурах, 'static, bounds, advanced

```rust
// Rust — компілятор НЕ ДОЗВОЛИТЬ!
fn get_value() -> &i32 {
    let x = 42;
    &x  // ❌ Error: `x` does not live long enough
}       // x dropped here

// Повідомлення компілятора:
// error[E0106]: missing lifetime specifier
// error: cannot return reference to local variable `x`
```

Rust гарантує: посилання завжди вказують на валідні дані!

```
fn main() {
    let r;                  // ┌──────────┐ 'a
                            //           |
    {                       //           |
        let x = 5;          // ┌──┐ 'b    |
        r = &x;             //    |       |
    }                       // └──┘       | x dropped, 'b ends
                            //           |
    println!("{}", r);      // ❌ Error  | r still in 'a, but points to dead 'b
}                           // └──────────┘
```

'a та 'b — lifetime параметри. r має 'a, але x має коротший 'b

```
fn main() {
    let x = 5;           // x створено
    let r = &x;          // r позичає x (borrow)

    println!("{}", r);   // ✓ OK — x ще живий

    // x живе до кінця main()
    // r живе до останнього використання
}

// Компілятор перевіряє:
// lifetime(r) ⊆ lifetime(x)  // r не переживає x
// ✓ Валідно!
```

```
// Lifetime параметр: апостроф + ім'я
'a    // Найпоширеніше ім'я
'b    // Друге
'c, 'd, 'e...  // Інші
'static  // Спеціальний — весь час роботи програми

// У типах посилань
&'a T        // Посилання з lifetime 'a на T
&'a mut T    // Мутабельне посилання з lifetime 'a

// У функціях
fn foo<'a>(x: &'a str) -> &'a str { ... }

// У структурах
struct Ref<'a> {
    data: &'a str,
}

// Конвенція: 'a, 'b, 'c для generic lifetimes
// 'static — глобальний lifetime
```

```rust
// Проблема: яке посилання повертаємо?
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
// ❌ Error: missing lifetime specifier
// Компілятор не знає: lifetime результату = x? y? інший?

// Рішення: explicit lifetime
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
// 'a — "generic lifetime parameter"
// Результат живе стільки, скільки живуть ОБА аргументи

// Використання
let s1 = String::from("long");
let s2 = String::from("longer");
let result = longest(&s1, &s2);  // ✓ OK
println!("{}", result);
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

let s1 = String::from("hello");        // ──────────────┐
{                                       //             |
    let s2 = String::from("world");    // ──────┐     |
    let result = longest(&s1, &s2);    //       |     |
    println!("{}", result);  // ✓ OK   //       |     |
}                                       // ──────┘     | s2 dropped
// result вже недоступний (lifetime 'a = min(s1, s2) = s2)

// Якби спробували:
// println!("{}", result);  // ❌ Error!
```

```
// Різні lifetimes для різних посилань
fn first_word<'a, 'b>(s: &'a str, prefix: &'b str) -> &'a str {
    // Повертаємо частину s, не prefix
    // Результат прив'язаний до 'a, не 'b
    if s.starts_with(prefix) {
        &s[prefix.len()..]
    } else {
        s
    }
}

let text = String::from("hello world");
let result;
{
    let prefix = String::from("hello ");
    result = first_word(&text, &prefix);
    // prefix може бути dropped тут
}  // prefix dropped

// result все ще валідний! (прив'язаний до text, не prefix)
println!("{}", result);  // "world"
```

Lifetime Elision Rules

Компілятор може автоматично вивести lifetimes у багатьох випадках.
Це називається "lifetime elision" (пропуск).

Три правила застосовуються послідовно:

Правило 1: Кожен вхідний параметр-посилання отримує свій lifetime

Правило 2: Якщо є тільки один вхідний lifetime — він присвоюється всім вихідним

Правило 3: Якщо є &self або &mut self — його lifetime присвоюється вихідним

Якщо після всіх правил lifetime не визначено — потрібна явна анотація!

```
// Правило 1: Кожен параметр-посилання отримує свій lifetime

// Ви пишете:
fn foo(x: &str) { }
fn bar(x: &str, y: &str) { }

// Компілятор додає:
fn foo<'a>(x: &'a str) { }
fn bar<'a, 'b>(x: &'a str, y: &'b str) { }

// Більше параметрів:
fn many(a: &i32, b: &str, c: &f64) { }
// Стає:
fn many<'a, 'b, 'c>(a: &'a i32, b: &'b str, c: &'c f64) { }

// Це лише перший крок!
// Далі застосовуються правила 2 та 3
```

```
// Правило 2: Якщо один input lifetime — він йде на всі outputs

// Ви пишете:
fn first_word(s: &str) -> &str { ... }

// Після правила 1:
fn first_word<'a>(s: &'a str) -> &str { ... }

// Після правила 2 (один input):
fn first_word<'a>(s: &'a str) -> &'a str { ... }
// ✓ Повністю виведено!

// Ще приклад:
fn get_first(items: &[i32]) -> Option<&i32> { ... }
// Стає:
fn get_first<'a>(items: &'a [i32]) -> Option<&'a i32> { ... }

// Правило 2 НЕ застосовується якщо > 1 input lifetime
fn longer(x: &str, y: &str) -> &str { ... }
// ❌ Два inputs — правило 2 не працює!
```

```
// Правило 3: У методах lifetime &self йде на всі outputs

struct Parser {
    input: String,
}

impl Parser {
    // Ви пишете:
    fn parse(&self) -> &str { ... }

    // Компілятор виводить:
    fn parse<'a>(&'a self) -> &'a str { ... }
    // ✓ Автоматично!
}

// Ще приклад:
impl MyStruct {
    fn get_name(&self) -> &str { &self.name }
    // Lifetime виведено: результат живе стільки ж, як &self

    fn combine(&self, other: &str) -> &str {
        // Тут все одно виведе 'self lifetime для результату
        // Але логічно може бути неправильно!
    }
}
```

```rust
// Кілька input lifetimes, результат — посилання
fn longest(x: &str, y: &str) -> &str {
    // ❌ Error!
    // Правило 1: fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str
    // Правило 2: не застосовується (два inputs)
    // Правило 3: не застосовується (немає self)
    // Lifetime результату невідомий!
}

// Рішення: явна анотація
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Інший варіант: різні lifetimes
fn choose_first<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    x  // Завжди повертаємо перший — lifetime 'a
}
```

```
struct Agent {
    id: u32,
    name: String,
    position: Position,
}
```

```
/// Знайти агента з найбільшим ID
/// Lifetime: результат живе стільки ж, як слайс агентів
fn find_max_id<'a>(agents: &'a [Agent]) -> Option<&'a Agent> {
    agents.iter().max_by_key(|a| a.id)
}

/// Знайти найближчого агента до позиції
fn find_nearest<'a>(agents: &'a [Agent], pos: &Position) -> Option<&'a Agent> {
    // pos не впливає на lifetime результату!
    agents.iter()
        .min_by(|a, b| {
            let da = a.position.distance_to(pos);
            let db = b.position.distance_to(pos);
            da.partial_cmp(&db).unwrap()
        })
}

let agents = vec![agent1, agent2, agent3];
let nearest = find_nearest(&agents, &target);  // ✓
```

```rust
/// Вибрати агента з кращою позицією
/// Обидва агенти мають однаковий lifetime
fn select_better_position<'a>(
    agent1: &'a Agent,
    agent2: &'a Agent,
    target: &Position,
) -> &'a Agent {
    let d1 = agent1.position.distance_to(target);
    let d2 = agent2.position.distance_to(target);

    if d1 < d2 { agent1 } else { agent2 }
}

// Використання
let a1 = Agent::new(1, "Alpha");
let a2 = Agent::new(2, "Beta");
let target = Position::new(100.0, 100.0);

let better = select_better_position(&a1, &a2, &target);
println!("Better positioned: {}", better.name);

// better валідний поки живі a1 та a2
```

```rust
/// Отримати всіх активних агентів
/// Повертаємо ітератор посилань
fn active_agents<'a>(
    agents: &'a [Agent]
) -> impl Iterator<Item = &'a Agent> {
    agents.iter().filter(|a| a.is_active())
}

/// Агенти в радіусі від позиції
fn agents_in_radius<'a>(
    agents: &'a [Agent],
    center: &Position,
    radius: f64,
) -> Vec<&'a Agent> {
    agents.iter()
        .filter(|a| a.position.distance_to(center) <= radius)
        .collect()
}

let agents = load_agents();
let center = Position::new(50.0, 50.0);

for agent in agents_in_radius(&agents, &center, 100.0) {
    println!("Agent {} in radius", agent.id);
}
// Посилання валідні поки живе agents
```

```
fn main() {
    let x = 5;              // ----------------------+ 'x
    let y = 10;             // --------------------+ | 'y
                            //                     | |
    let r = pick(&x, &y);   // ----------+ 'r     | |
                            //           |         | |
    println!("{}", r);      //           |         | |
                            // ----------+         | |
    println!("{}", y);      //                     | |
                            // --------------------+ |
    println!("{}", x);      //                       |
}                           // ----------------------+

fn pick<'a>(x: &'a i32, y: &'a i32) -> &'a i32 {
    if *x > *y { x } else { y }
}

// 'r ⊆ 'a ⊆ min('x, 'y) = 'y
// r валідний поки живі і x, і y
```

```rust
// До Rust 2018: lexical lifetimes (до кінця scope)
// Rust 2018+: non-lexical lifetimes (до останнього використання)

fn main() {
    let mut v = vec![1, 2, 3];

    let first = &v[0];      // immutable borrow
    println!("{}", first);  // останнє використання first

    // Старий Rust: first живе до }, не можна мутувати
    // NLL: first "мертвий" після println, можна мутувати!

    v.push(4);  // ✓ OK з NLL
    println!("{:?}", v);
}

// NLL робить lifetimes "розумнішими"
// borrow закінчується при останньому використанні,
// а не в кінці блоку
```

NLL увімкнено за замовчуванням з Rust 2018 edition

Приклади Lifetime Elision

Ви пишете	Компілятор виводить
fn f(x: &str)	fn f<'a>(x: &'a str)
fn f(x: &str) -> &str	fn f<'a>(x: &'a str) -> &'a str
fn f(&self) -> &str	fn f<'a>(&'a self) -> &'a str
fn f(&self, s: &str) -> &str	fn f<'a>(&'a self, s: &str) -> &'a str
fn f(x: &str, y: &str)	fn f<'a, 'b>(x: &'a str, y: &'b str)
fn f(x: &str, y: &str) -> &str	❌ Error — потрібна анотація!

Останній випадок: два inputs, один output — компілятор не знає який lifetime

Типові помилки з lifetimes

```
let r;
{ let x = 5; r = &x; }
println!("{}", r); // ❌
```

✓ Дані живуть довше

```
let x = 5;
let r = &x;
println!("{}", r); // ✓
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value,
          but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

Компілятор Rust дуже допомагає:

- Показує де проблема
- Пояснює чому
- Пропонує рішення (help: consider...)

Підсумок: Частина 1

Lifetimes — механізм відстеження часу життя посилань:
• Гарантують відсутність dangling references
• Перевіряються на етапі компіляції
• Annotations не змінюють lifetime — лише описують зв'язки

Lifetime Elision — автоматичне виведення:
• Правило 1: кожен input отримує свій lifetime
• Правило 2: один input → він же на output
• Правило 3: &self → його lifetime на output

Якщо elision не працює — потрібна явна анотація <'a>

🤖 MAC: функції пошуку/фільтрації агентів з посиланнями

→ Частина 2: Lifetimes у структурах, 'static, bounds

Лекція 11 (продовження)

Lifetimes у структурах та 'static

Структури з посиланнями та lifetime bounds

struct<'a> • 'static • T: 'a • HRTB • subtyping

🤖 Приклади: AgentView, MissionRef, StateSnapshot

Частина 2: Структури, 'static, bounds

О. С. Бичков • 2025

План лекції (Частина 2)

1. Lifetimes у структурах
2. Методи структур з lifetimes
3. 'static lifetime
4. String literals та 'static
5. 'static у generics
6. T: 'static — що це означає
7. Lifetime bounds
8. T: 'a bounds
9. Lifetime subtyping
10. Higher-Ranked Trait Bounds
11. 🤖 AgentView pattern
12. 🤖 MissionRef структура
13. 🤖 StateSnapshot
14. 🤖 Ітератор по агентах
15. Практичні поради
16. Коли уникати lifetimes

```rust
// Структура з посиланнями ПОТРЕБУЄ lifetime параметр
struct ImportantExcerpt<'a> {
    part: &'a str,
}

// Структура не може пережити дані, на які посилається
let novel = String::from("Call me Ishmael. Some years ago...");
let first_sentence = novel.split('.').next().unwrap();

let excerpt = ImportantExcerpt {
    part: first_sentence,  // Посилання на частину novel
};

println!("{}", excerpt.part);  // ✓ OK

// excerpt живе не довше за novel
// drop(novel); // Якби тут - excerpt став би dangling
```

Lifetime параметр структури = я не переживу дані, на які посилалось

```rust
// Різні поля можуть мати різні lifetimes
struct TwoRefs<'a, 'b> {
    first: &'a str,
    second: &'b str,
}

// Приклад використання
let s1 = String::from("hello");
let result;
{
    let s2 = String::from("world");
    let two = TwoRefs {
        first: &s1,
        second: &s2,
    };
    result = two.first;  // first має lifetime 'a (s1)
}  // s2 dropped, але result валідний!

println!("{}", result);  // "hello" — все ще валідний

// Якби result = two.second — компілятор не дозволив би
// бо second прив'язаний до s2, який вже dropped
```

```rust
struct Excerpt<'a> {
    text: &'a str,
}

// impl теж потребує lifetime параметр
impl<'a> Excerpt<'a> {
    // Конструктор
    fn new(text: &'a str) -> Self {
        Excerpt { text }
    }

    // Метод — elision працює! (правило 3)
    fn first_word(&self) -> &str {
        self.text.split_whitespace().next().unwrap_or("")
    }

    // Явний lifetime
    fn announce_and_return(&self, announcement: &str) -> &'a str {
        println!("Attention: {}", announcement);
        self.text  // Повертаємо з lifetime 'a, не lifetime &self
    }
}

let text = String::from("hello world");
let ex = Excerpt::new(&text);
println!("{}", ex.first_word());  // "hello"
```

```
// String literals — 'static
let s: &'static str = "hello, world";
// Літерал вбудований у бінарний файл — живе "вічно"

// Константи — 'static
static GREETING: &str = "Hello!";
const PI: f64 = 3.14159;  // Теж 'static

// Box::leak — створити 'static з heap даних
let leaked: &'static String = Box::leak(Box::new(String::from("leaked")));
// Пам'ять ніколи не звільниться!
```

```
// ⚠️ 'static не означає "незмінний"
// static mut існує (unsafe)
```

```
// T: 'static означає:
// "T може бути owned типом АБО &'static посиланням"
// НЕ означає "T є &'static"!

use std::thread;

// thread::spawn вимагає T: 'static
fn spawn_thread<T: 'static + Send>(data: T) {
    thread::spawn(move || {
        // data може жити скільки завгодно
        println!("Got data");
    });
}

// Owned типи — T: 'static ✓
spawn_thread(String::from("hello"));  // ✓ String: 'static
spawn_thread(42i32);                   // ✓ i32: 'static
spawn_thread(vec![1, 2, 3]);           // ✓ Vec<i32>: 'static

// Посилання — НЕ 'static (якщо не &'static)
let s = String::from("temp");
// spawn_thread(&s);  // ❌ &String не 'static
```

```
struct HasRef<'a> {
    data: &'a str,
}

// HasRef<'a> НЕ: 'static (бо містить 'a посилання)
// HasRef<'static>: 'static (бо посилання 'static)

fn needs_static<T: 'static>(x: T) {}

needs_static(42);                    // ✓ i32: 'static
needs_static(String::from("hi"));    // ✓ String: 'static
needs_static("literal");             // ✓ &'static str

let s = String::from("local");
// needs_static(&s);  // ❌ &String не 'static

let has_static = HasRef { data: "literal" };
needs_static(has_static);  // ✓ HasRef<'static>
```

```rust
// T: 'a означає "T живе принаймні 'a"
// Якщо T містить посилання — вони мають жити >= 'a

struct Wrapper<'a, T: 'a> {
    value: &'a T,
}

// Тепер T має жити принаймні 'a
impl<'a, T: 'a> Wrapper<'a, T> {
    fn new(value: &'a T) -> Self {
        Wrapper { value }
    }
}

// Приклад
let num = 42;
let wrapper = Wrapper::new(&num);
// wrapper валідний поки живе num

// Без T: 'a компілятор видав би помилку:
// "the parameter type `T` may not live long enough"
```

```rust
use std::fmt::Debug;

// Lifetime + trait bounds
fn print_ref<'a, T: Debug + 'a>(x: &'a T) {
    println!("{:?}", x);
}

// where clause для читабельності
fn complex<'a, 'b, T, U>(x: &'a T, y: &'b U) -> &'a T
where
    T: Debug + Clone + 'a,
    U: Debug + 'b,
    'b: 'a,  // 'b >= 'a (b outlives a)
{
    println!("{:?}, {:?}", x, y);
    x
}

// 'b: 'a означає 'b >= 'a ("b outlives a")
// Тобто все, що живе 'b, живе і 'a

// Практичний приклад
fn longest_with_announcement<'a, T: Display>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str {
    println!("Announcement: {}", ann);
    if x.len() > y.len() { x } else { y }
}
```

```
// Довший lifetime є subtype коротшого!
// 'long: 'short означає 'long >= 'short

fn example<'long, 'short>(long_ref: &'long str, short_ref: &'short str)
where
    'long: 'short,  // 'long outlives 'short
{
    // Можемо використати &'long там, де очікується &'short
    let s: &'short str = long_ref;  // ✓ OK! coercion
}

// Інтуїтивно: якщо щось живе довше — його можна використати
// там, де потрібно щось, що живе коротше

// 'static — найдовший lifetime
// 'static: 'a для будь-якого 'a

fn takes_short<'a>(s: &'a str) {}

let static_str: &'static str = "hello";
takes_short(static_str);  // ✓ 'static coerces to 'a
```

Higher-Ranked Trait Bounds (HRTB)

```rust
// HRTB: for<'a> — "для будь-якого lifetime 'a"

// Проблема: функція що приймає closure з посиланням
fn call_with_ref<F>(f: F)
where
    F: Fn(&i32) -> i32,  // Який lifetime у &i32?
{
    let x = 42;
    f(&x);
}

// Rust автоматично розуміє це як:
fn call_with_ref<F>(f: F)
where
    F: for<'a> Fn(&'a i32) -> i32,  // Для БУДЬ-ЯКОГО 'a
{
    let x = 42;
    f(&x);  // Працює з локальним x
}

// HRTB рідко потрібні явно — компілятор виводить
call_with_ref(|x| x + 1);  // ✓ Автоматично
```

for<'a> — "ця функція працює з посиланням будь-якого lifetime"

```rust
/// View на стан агента (без копіювання)
struct AgentView<'a> {
    id: u32,
    name: &'a str,
    position: &'a Position,
    battery: u8,
}

impl Agent {
    /// Створити view на агента
    fn view(&self) -> AgentView<'_> {
        AgentView {
            id: self.id,
            name: &self.name,
            position: &self.position,
            battery: self.battery,
        }
    }
}

// Використання — ефективне, без копіювання
fn print_agent(view: AgentView<'_>) {
    println!("Agent {}: {} at {:?}, battery {}%",
        view.id, view.name, view.position, view.battery);
}

let agent = Agent::new(1, "Alpha");
print_agent(agent.view());  // View живе поки живе agent
```

MAS Mission Reference

```rust
/// Посилання на активну місію
struct MissionRef<'a> {
    mission: &'a Mission,
    assigned_agents: &'a [AgentId],
    progress: &'a MissionProgress,
}

impl<'a> MissionRef<'a> {
    /// Перевірити чи агент призначений
    fn is_agent_assigned(&self, agent_id: AgentId) -> bool {
        self.assigned_agents.contains(&agent_id)
    }

    /// Отримати прогрес у відсотках
    fn progress_percent(&self) -> f32 {
        self.progress.completed as f32 / self.progress.total as f32 * 100.0
    }

    /// Отримати назву місії
    fn name(&self) -> &'a str {
        &self.mission.name
    }
}

// Функція, що повертає MissionRef
fn get_active_mission<'a>(state: &'a SwarmState) -> Option<MissionRef<'a>> {
    state.active_mission.as_ref().map(|m| MissionRef {
        mission: m,
        assigned_agents: &state.assigned_agents,
        progress: &state.mission_progress,
```

```rust
/// Знімок стану системи (zero-copy)
struct StateSnapshot<'a> {
    timestamp: u64,
    agents: &'a [Agent],
    active_missions: &'a [Mission],
    world_map: &'a WorldMap,
}

impl<'a> StateSnapshot<'a> {
    /// Створити snapshot поточного стану
    fn capture(state: &'a SystemState) -> Self {
        StateSnapshot {
            timestamp: current_time(),
            agents: &state.agents,
            active_missions: &state.missions,
            world_map: &state.map,
        }
    }

    /// Аналіз стану (read-only)
    fn analyze(&self) -> AnalysisReport {
        AnalysisReport {
            agent_count: self.agents.len(),
            active_agents: self.agents.iter().filter(|a| a.is_active()).count(),
            mission_count: self.active_missions.len(),
            coverage: self.calculate_coverage(),
        }
    }
}
```

```
/// Ітератор по активних агентах
struct ActiveAgentsIter<'a> {
    agents: &'a [Agent],
    index: usize,
}

impl<'a> ActiveAgentsIter<'a> {
    fn new(agents: &'a [Agent]) -> Self {
        ActiveAgentsIter { agents, index: 0 }
    }
}

impl<'a> Iterator for ActiveAgentsIter<'a> {
    type Item = &'a Agent;

    fn next(&mut self) -> Option<Self::Item> {
        while self.index < self.agents.len() {
            let agent = &self.agents[self.index];
            self.index += 1;
            if agent.is_active() {
                return Some(agent);
            }
        }
        None
    }
}

// Використання
for agent in ActiveAgentsIter::new(&agents) {
    println!("Active: {}", agent.name);
```

🕹️ MAS Builder — модифікація

```rust
/// Builder для команди з посиланнями на ресурси
struct CommandBuilder<'a> {
    target: Option<&'a Position>,
    agents: Vec<&'a Agent>,
    priority: u8,
}

impl<'a> CommandBuilder<'a> {
    fn new() -> Self {
        CommandBuilder {
            target: None,
            agents: Vec::new(),
            priority: 5,
        }
    }

    fn target(mut self, pos: &'a Position) -> Self {
        self.target = Some(pos);
        self
    }

    fn add_agent(mut self, agent: &'a Agent) -> Self {
        self.agents.push(agent);
        self
    }

    fn priority(mut self, p: u8) -> Self {
        self.priority = p;
        self
    }
```

```
// Замість складних lifetimes:
struct Complex<'a, 'b> { a: &'a str, b: &'b str }

// Часто простіше owned:
struct Simple { a: String, b: String }
```

```rust
// '_ — "placeholder" lifetime, компілятор виведе сам

struct Wrapper<'a> {
    data: &'a str,
}

impl Wrapper<'_> {  // '_ замість явного 'a
    fn len(&self) -> usize {
        self.data.len()
    }
}

// Корисно коли lifetime очевидний
fn first_word(s: &str) -> &str {
    // ...
}

// Те саме з '_:
fn first_word(s: &'_ str) -> &'_ str {
    // ...
}

// У типах:
let x: Wrapper<'_> = Wrapper { data: "hello" };

// '_ каже: "тут є lifetime, але я не хочу називати його"
```

```rust
// Closures можуть захоплювати посилання
fn main() {
    let data = vec![1, 2, 3];

    // Closure захоплює &data
    let print_data = || {
        println!("{:?}", data);  // Borrow data
    };

    print_data();  // ✓ OK
    print_data();  // ✓ OK

    // data все ще доступна
    println!("{:?}", data);
}

// move closure — ownership, не borrow
let owned_closure = move || {
    println!("{:?}", data);  // data moved into closure
};
// data більше недоступна тут

// Для threads потрібен move (дані мають бути 'static)
std::thread::spawn(move || {
    // closure володіє своїми даними
});
```

Порівняння підходів

Підхід	Lifetime	Copy/Clone	Коли
&T (borrow)	Так	Ні	Read-only, temp
&mut T	Так	Ні	Modify, temp
T (owned)	Ні	Move	Transfer ownership
Clone	Ні	Так	Independent copy
Rc<T>	Ні	Ні	Shared ownership
Arc<T>	Ні	Ні	Thread-safe shared
View struct	Так	Ні	Zero-copy view

View з lifetime — ефективно, але обмежено scope

Практичні поради

✓ Починайте без lifetime annotations — нехай elision працює
✓ Додавайте annotations тільки коли компілятор вимагає
✓ Читайте повідомлення компілятора — він підказує
✓ Використовуйте '_ коли lifetime очевидний
✓ View patterns для ефективного read-only доступу

❌ Не ускладнюйте lifetime annotations без потреби
❌ Не борітеся з borrow checker — можливо owned краще
❌ Не використовуйте 'static без необхідності

🤖 Для MAC:
• AgentView<'a> — для функцій аналізу
• StateSnapshot<'a> — для read-only операцій
• Owned типи — для даних, що передаються між потоками

Типові помилки

```
// 'static не робить owned!
let s: &'static str = "hi";
// s — все ще посилання
```

✓ Owned для threads

```
// Для threads — owned типи
let s = String::from("hi");
thread::spawn(move || { ...
```

Підсумок лекції

Lifetimes у структурах:
• struct Name<'a> { field: &'a T }
• Структура не переживає дані, на які посилається

'static lifetime:
• Дані живуть "вічно" (літерали, константи)
• T: 'static — T owned або містить 'static refs

Lifetime bounds:
• T: 'a — T живе принаймні 'a
• 'a: 'b — 'a >= 'b (outlives)

🤖 MAC patterns:
• AgentView<'a> — ефективний view
• StateSnapshot<'a> — zero-copy snapshot
• Iterator з lifetime

→ Наступна лекція: Threads — багатопотоковість

Завдання для самостійної роботи

1. Excerpt: Структура Excerpt<'a> з &'a str.
   Методи: new, first_word, word_count.

2. Longest: Функція longest_line<'a>(&'a [&'a str]) -> &'a str
   Повертає найдовший рядок.

3. AgentView: Реалізуйте View pattern:
   - AgentView<'a> з посиланнями на поля Agent
   - Метод view(&self) для Agent
   - Функції аналізу що приймають AgentView

4. StateSnapshot: Структура з посиланнями на:
   - agents: &'a [Agent]
   - missions: &'a [Mission]
   - Методи analyze, summary

5. Iterator: Створіть FilteredAgentsIter<'a> з предикатом.

Дякую за увагу!

Lifetimes • 'static • bounds • View patterns

Питання?