Лекція 9

Box<T> та Rc<T>: Розумні вказівники

Heap allocation та shared ownership

Box • Rc • Weak • Deref • Drop

🤖 Приклади: дерева рішень, графи агентів, спільні ресурси

Частина 1: Box<T> — heap allocation

О. С. Бичков • 2025

План лекції (Частина 1: Box)

1. Stack vs Heap
2. Що таке Smart Pointer?
3. Box<T> — базове використання
4. Навіщо потрібен Box?
5. Рекурсивні типи
6. Box для trait objects
7. Deref trait
8. Deref coercion
9. Drop trait
10. std::mem::drop
11. Box::leak
12. 🤖 Дерево рішень агента
13. 🤖 Поліморфні команди
14. 🤖 Recursive state machine
15. Порівняння: Box vs інші

Частина 2: Rc<T>, Weak<T>, RefCell<T>, Cell<T>

Stack vs Heap: нагадування

Stack (стек)

- Швидке виділення/звільнення
- Фіксований розмір на компіляції
- LIFO — Last In, First Out
- Автоматичне звільнення
- Локальні змінні, параметри

Типи: i32, f64, [T; N], struct
без Box/Rc/Vec

Heap (купа)

- Повільніше виділення
- Динамічний розмір
- Довільний порядок
- Потребує явного керування
- Дані, що живуть довше scope

Типи: Box<T>, Rc<T>, Vec<T>,
String, HashMap<K,V>

Rust: ownership система керує heap автоматично — без GC!

Що таке Smart Pointer?

Smart Pointer — структура, що поводиться як вказівник,
але має додаткові метадані та можливості

В Rust smart pointers:
- Володіють даними (ownership)
- Реалізують Deref — поведінка як посилання

Box<T>

Heap allocation
Єдиний власник

Rc<T>

Reference counting
Спільне володіння

RefCell<T>

Interior mutability
Runtime borrow check

String, Vec<T> — теж smart pointers! Володіють heap даними.

```rust
// Box<T> — розміщує значення на heap
let x = 5;                    // i32 на stack
let boxed = Box::new(5);      // i32 на heap, Box на stack

// Box поводиться як звичайне значення
println!("boxed = {}", boxed);  // 5 (Deref автоматично)

// Deref дозволяє доступ до внутрішнього значення
let y = *boxed;  // Розпакування (dereference)
println!("y = {}", y);  // 5

// Box володіє даними
let b1 = Box::new(String::from("hello"));
let b2 = b1;  // b1 moved!
// println!("{}", b1);  // ❌ Error: b1 moved

// При виході з scope — дані на heap звільняються
{
    let temp = Box::new(vec![1, 2, 3]);
}  // temp і Vec звільнені тут
```

Навіщо потрібен Box<T>?

```
struct BigData {
    data: [u8; 1_000_000], // 1 MB!
}
let x = BigData { ... };
// Stack overflow ризик!
```

З Box — на heap

```
let x = Box::new(BigData { ... });
// Box на stack: ~8 bytes
// BigData на heap: 1 MB
```

```
// Linked List — класичний рекурсивний тип
enum List {
    Cons(i32, List),  // ❌ Error: infinite size!
    Nil,
}

// Компілятор:
// "recursive type `List` has infinite size"
// "help: insert some indirection (e.g., a `Box`, `Rc`, or `&`)"
```

Чому?

Rust має знати розмір кожного типу на етапі компіляції.

```
Cons(i32, List) = 4 bytes + sizeof(List)
                = 4 + (4 + sizeof(List))
                = 4 + (4 + (4 + sizeof(List)))
                = ... нескінченно!
```

```
// Box має фіксований розмір (pointer = 8 bytes на 64-bit)
enum List {
    Cons(i32, Box<List>),  // ✓ 4 + 8 = 12 bytes
    Nil,
}

use List::{Cons, Nil};

let list = Cons(1,
    Box::new(Cons(2,
        Box::new(Cons(3,
            Box::new(Nil))))));

// Візуально:
// Stack: [1, ptr] -> Heap: [2, ptr] -> Heap: [3, ptr] -> Heap: [Nil]

// Методи для роботи зі списком
impl List {
    fn len(&self) -> usize {
        match self {
            Nil => 0,
            Cons(_, next) => 1 + next.len(),
        }
    }
}
```

```rust
// Бінарне дерево — теж рекурсивний тип
struct TreeNode<T> {
    value: T,
    left: Option<Box<TreeNode<T>>>,
    right: Option<Box<TreeNode<T>>>,
}

impl<T> TreeNode<T> {
    fn new(value: T) -> Self {
        TreeNode { value, left: None, right: None }
    }

    fn with_children(value: T, left: TreeNode<T>, right: TreeNode<T>) -> Self {
        TreeNode {
            value,
            left: Some(Box::new(left)),
            right: Some(Box::new(right)),
        }
    }
}

// Побудова дерева
let tree = TreeNode::with_children(
    10,
    TreeNode::new(5),
    TreeNode::with_children(15, TreeNode::new(12), TreeNode::new(20))
);
```

```rust
trait Animal {
    fn speak(&self) -> String;
}

struct Dog;
struct Cat;

impl Animal for Dog {
    fn speak(&self) -> String { "Woof!".to_string() }
}

impl Animal for Cat {
    fn speak(&self) -> String { "Meow!".to_string() }
}

// dyn Animal — невідомий розмір, потрібен Box
fn get_animal(is_dog: bool) -> Box<dyn Animal> {
    if is_dog {
        Box::new(Dog)
    } else {
        Box::new(Cat)
    }
}

// Гетерогенна колекція
let animals: Vec<Box<dyn Animal>> = vec![
    Box::new(Dog),
    Box::new(Cat),
];
```

```rust
use std::ops::Deref;

// Deref дозволяє використовувати * для розпакування
impl<T> Deref for Box<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        // Повертає посилання на внутрішнє значення
    }
}

// Приклад використання
let x = Box::new(5);

// * викликає deref()
assert_eq!(*x, 5);

// Під капотом:
// *x еквівалентно *(x.deref())

// Власний smart pointer
struct MyBox<T>(T);

impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.0
    }
}
```

```rust
// Deref coercion — Rust автоматично викликає deref()
// коли типи не збігаються

fn hello(name: &str) {
    println!("Hello, {}!", name);
}

let boxed_string = Box::new(String::from("World"));

// Без deref coercion:
hello(&(*boxed_string)[..]);

// З deref coercion (автоматично):
hello(&boxed_string);  // Box<String> -> &String -> &str

// Ланцюжок coercion:
// &Box<String>
//   -> &String (Box::deref)
//   -> &str    (String::deref)

// Працює для:
// &T -> &U якщо T: Deref<Target=U>
// &mut T -> &mut U якщо T: DerefMut<Target=U>
// &mut T -> &U якщо T: Deref<Target=U>
```

```rust
// Drop викликається автоматично при виході зі scope
trait Drop {
    fn drop(&mut self);
}

struct CustomBox<T> {
    data: T,
}

impl<T> Drop for CustomBox<T> {
    fn drop(&mut self) {
        println!("CustomBox dropped!");
    }
}

fn main() {
    let _a = CustomBox { data: String::from("hello") };
    let _b = CustomBox { data: 42 };
    println!("Created boxes");
}  // b dropped, потім a dropped (зворотній порядок!)

// Вивід:
// Created boxes
// CustomBox dropped!
// CustomBox dropped!

// Box<T> автоматично звільняє heap пам'ять у drop()
```

```rust
// Не можна викликати drop() напряму!
let x = Box::new(5);
// x.drop();  // ❌ Error: explicit destructor calls not allowed

// Використовуйте std::mem::drop()
let x = Box::new(5);
drop(x);  // ✓ x знищено
// println!("{}", x);  // ❌ Error: x moved

// Практичне застосування
struct Lock {
    name: String,
}

impl Drop for Lock {
    fn drop(&mut self) {
        println!("Lock '{}' released", self.name);
    }
}

fn main() {
    let lock = Lock { name: "mutex".to_string() };

    // Критична секція...

    drop(lock);  // Явно звільняємо lock

    // Тепер інші можуть захопити lock
}
```

```rust
// Box::leak "витікає" пам'ять, повертаючи &'static mut T
// Корисно для глобальних даних

let boxed = Box::new(String::from("I live forever"));
let leaked: &'static mut String = Box::leak(boxed);

// leaked має 'static lifetime — живе до кінця програми
println!("{}", leaked);  // Працює будь-де

// Практичне застосування: lazy static
static mut CONFIG: Option<&'static Config> = None;

fn init_config() {
    let config = Box::new(Config::load());
    unsafe {
        CONFIG = Some(Box::leak(config));
    }
}

// Або краще — використовуйте once_cell / lazy_static
use once_cell::sync::Lazy;

static CONFIG: Lazy<Config> = Lazy::new(|| {
    Config::load()
});
```

```rust
/// Вузол дерева рішень
enum DecisionNode {
    /// Умова — перевіряє стан і йде в одну з гілок
    Condition {
        check: Box<dyn Fn(&AgentState) -> bool>,
        if_true: Box<DecisionNode>,
        if_false: Box<DecisionNode>,
    },
    /// Дія — кінцевий вузол
    Action(AgentAction),
}

impl DecisionNode {
    fn evaluate(&self, state: &AgentState) -> AgentAction {
        match self {
            DecisionNode::Condition { check, if_true, if_false } => {
                if check(state) {
                    if_true.evaluate(state)
                } else {
                    if_false.evaluate(state)
                }
            }
            DecisionNode::Action(action) => action.clone(),
        }
    }
}

// Побудова дерева
let tree = DecisionNode::Condition {
    check: Box::new(|s| s.battery < 20),
```

🕹️ [illegible]

```rust
/// Behavior Tree — популярна модель AI
enum BehaviorNode {
    /// Виконує послідовно, зупиняється на першій невдачі
    Sequence(Vec<Box<BehaviorNode>>),
    /// Виконує до першого успіху
    Selector(Vec<Box<BehaviorNode>>),
    /// Умова
    Condition(Box<dyn Fn(&Drone) -> bool>),
    /// Дія
    Action(Box<dyn Fn(&mut Drone) -> BehaviorStatus>),
}

enum BehaviorStatus { Success, Failure, Running }

impl BehaviorNode {
    fn tick(&self, drone: &mut Drone) -> BehaviorStatus {
        match self {
            BehaviorNode::Sequence(children) => {
                for child in children {
                    match child.tick(drone) {
                        BehaviorStatus::Failure => return BehaviorStatus::Failure,
                        BehaviorStatus::Running => return BehaviorStatus::Running,
                        BehaviorStatus::Success => continue,
                    }
                }
                BehaviorStatus::Success
            }
            // ...
        }
    }
```

```rust
/// Trait для команд
trait Command {
    fn execute(&self, drone: &mut Drone) -> Result<(), CommandError>;
    fn undo(&self, drone: &mut Drone) -> Result<(), CommandError>;
}

struct MoveCommand { destination: Position }
struct RotateCommand { angle: f64 }
struct FireCommand { target: TargetId }

impl Command for MoveCommand {
    fn execute(&self, drone: &mut Drone) -> Result<(), CommandError> {
        drone.move_to(self.destination)
    }
    fn undo(&self, drone: &mut Drone) -> Result<(), CommandError> {
        drone.move_to(drone.previous_position)
    }
}

/// Черга команд з можливістю undo
struct CommandHistory {
    executed: Vec<Box<dyn Command>>,
}

impl CommandHistory {
    fn execute(&mut self, cmd: Box<dyn Command>, drone: &mut Drone) {
        if cmd.execute(drone).is_ok() {
            self.executed.push(cmd);
        }
    }
```

```rust
/// Стан агента як рекурсивний тип
enum AgentState {
    Idle,
    Moving {
        target: Position,
        next_state: Box<AgentState>,  // Куди перейти після руху
    },
    Scanning {
        area: Area,
        on_found: Box<AgentState>,    // Якщо знайшли
        on_empty: Box<AgentState>,    // Якщо порожньо
    },
    Attacking {
        target: TargetId,
        on_success: Box<AgentState>,
        on_failure: Box<AgentState>,
    },
    Returning,
}

impl AgentState {
    /// Створення ланцюжка станів
    fn patrol_and_return() -> Self {
        AgentState::Moving {
            target: Position::new(100.0, 100.0),
            next_state: Box::new(AgentState::Scanning {
                area: Area::default(),
                on_found: Box::new(AgentState::Attacking { /* ... */ }),
                on_empty: Box::new(AgentState::Returning),
            }),
```

```rust
/// Trait для плагінів агента
trait AgentPlugin: Send + Sync {
    fn name(&self) -> &str;
    fn on_tick(&mut self, agent: &mut AgentData);
    fn on_message(&mut self, agent: &mut AgentData, msg: &Message);
}

/// Агент з динамічними плагінами
struct Agent {
    data: AgentData,
    plugins: Vec<Box<dyn AgentPlugin>>,
}

impl Agent {
    fn add_plugin(&mut self, plugin: Box<dyn AgentPlugin>) {
        println!("Adding plugin: {}", plugin.name());
        self.plugins.push(plugin);
    }

    fn tick(&mut self) {
        for plugin in &mut self.plugins {
            plugin.on_tick(&mut self.data);
        }
    }
}

// Реєстрація плагінів
let mut agent = Agent::new();
agent.add_plugin(Box::new(NavigationPlugin::new()));
agent.add_plugin(Box::new(CommunicationPlugin::new()));
```

Основні методи Box<T>

Метод	Опис	Приклад
Box::new(v)	Створити Box	Box::new(42)
*boxed	Deref — доступ до значення	let x = *boxed;
Box::into_raw(b)	Конвертувати в raw pointer	let ptr = Box::into_raw(b);
Box::from_raw(p)	Створити з raw pointer	unsafe { Box::from_raw(p) }
Box::leak(b)	"Витекти" в 'static	let s: &'static = Box::leak(b);
Box::pin(v)	Створити Pin<Box<T>>	Box::pin(future)

Box реалізує Deref, DerefMut, Drop, Clone (якщо T: Clone), Debug (якщо T: Debug)

Порівняння: Box vs інші типи

Тип	Ownership	Mutability	Thread-safe
Box<T>	Єдиний власник	Звичайна	✓ (якщо T: Send)
Rc<T>	Спільне (лічильник)	Тільки read	❌ (single-thread)
Arc<T>	Спільне (atomic)	Тільки read	✓
RefCell<T>	Єдиний	Runtime borrow	❌
Mutex<T>	Спільне	Lock-based	✓

Коли використовувати Box<T>:
- Рекурсивні типи (дерева, списки)
- Trait objects (dyn Trait)
- Великі дані на heap
- Передача ownership без копіювання

Типові помилки з Box

```
enum List { Cons(i32, List) }
// ❌ Infinite size!
```

✓ Правильно

```
enum List { Cons(i32, Box<List>) }
// ✓ Fixed size!
```

Підсумок: Box<T>

Box<T> — найпростіший smart pointer:
- Розміщує дані на heap
- Єдиний власник (ownership)
- Фіксований розмір (pointer)
- Автоматичне звільнення (Drop)

Коли використовувати:
- Рекурсивні типи
- Box<dyn Trait>
- Великі структури
- Transfer ownership

Важливі traits:
- Deref — поведінка як &T
- Drop — очищення ресурсів

→ Частина 2: Rc<T>, Weak<T>, RefCell<T>

Лекція 9 (продовження)

Rc<T>, Weak<T>, RefCell<T>

Спільне володіння та interior mutability

Rc • Weak • RefCell • Cell • циклічні посилання

🤖 Приклади: спільна карта, графи агентів, кеш

Частина 2: Shared ownership

О. С. Бичков • 2025

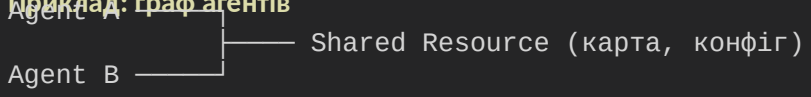
План лекції (Частина 2)

1. Проблема shared ownership
2. Rc<T> — Reference Counting
3. Rc::clone vs .clone()
4. Rc::strong_count
5. Проблема циклів
6. Weak<T> — слабкі посилання
7. Weak::upgrade
8. RefCell<T> — interior mutability
9. borrow() та borrow_mut()
10. Rc<RefCell<T>> pattern
11. Cell<T> — для Copy типів
12. 🤖 Спільна карта світу
13. 🤖 Граф комунікації агентів
14. 🤖 Observer pattern
15. 🤖 Кеш з weak references
16. Порівняння smart pointers

```
// Ownership каже: один власник
let data = vec![1, 2, 3];
let a = data;
let b = data;  // ❌ Error: data moved!

// Але іноді потрібно спільне володіння:
// • Граф — вузол має кілька батьків
// • Кеш — кілька компонентів читають
// • Observer — кілька підписників
```

Приклад: граф агентів



Хто володіє Shared Resource?
A? B? Обидва? Ніхто?

Рішення: Rc<T> — лічильник посилань

```rust
use std::rc::Rc;

// Rc — спільне володіння через підрахунок посилань
let data = Rc::new(vec![1, 2, 3]);

// Rc::clone — створює новий Rc, збільшує лічильник
let a = Rc::clone(&data);  // count = 2
let b = Rc::clone(&data);  // count = 3

// Всі вказують на ті самі дані!
println!("{:?}", data);  // [1, 2, 3]
println!("{:?}", a);     // [1, 2, 3]
println!("{:?}", b);     // [1, 2, 3]

// Дані звільняються коли count = 0
drop(a);  // count = 2
drop(b);  // count = 1
// data виходить зі scope → count = 0 → Дані звільнені
```

⚠️ Rc дані лише для однопоточного коду. Для потоків — Arc.

```rust
use std::rc::Rc;

let original = Rc::new(String::from("hello"));

// Rc::clone — збільшує лічильник, НЕ копіює дані
let reference = Rc::clone(&original);  // O(1), cheap

// .clone() на Rc — те саме!
let reference2 = original.clone();  // Теж O(1)

// Конвенція: використовуйте Rc::clone для ясності
let good = Rc::clone(&original);  // Явно: це Rc clone
let confusing = original.clone(); // Можна сплутати з deep clone

// Різниця з глибоким клонуванням:
let data = Rc::new(vec![1, 2, 3]);
let rc_clone = Rc::clone(&data);      // Той самий Vec!
let deep_clone = (*data).clone();      // Новий Vec!

// Перевірка
assert!(Rc::ptr_eq(&data, &rc_clone));   // true — той самий
// deep_clone — інший Vec
```

```rust
use std::rc::Rc;

let data = Rc::new(42);
println!("count: {}", Rc::strong_count(&data));  // 1

let a = Rc::clone(&data);
println!("count: {}", Rc::strong_count(&data));  // 2

{
    let b = Rc::clone(&data);
    println!("count: {}", Rc::strong_count(&data));  // 3
}  // b dropped

println!("count: {}", Rc::strong_count(&data));  // 2

drop(a);
println!("count: {}", Rc::strong_count(&data));  // 1

// Коли strong_count = 0 — дані звільняються

// Корисно для дебагу
fn debug_rc<T>(rc: &Rc<T>, name: &str) {
    println!("{}: {} references", name, Rc::strong_count(rc));
}
```

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Вузол, що посилається на інші вузли
struct Node {
    value: i32,
    next: Option<Rc<RefCell<Node>>>,
}

let a = Rc::new(RefCell::new(Node { value: 1, next: None }));
let b = Rc::new(RefCell::new(Node { value: 2, next: None }));

// Створюємо цикл: a -> b -> a
a.borrow_mut().next = Some(Rc::clone(&b));
b.borrow_mut().next = Some(Rc::clone(&a));  // ЦИКЛ!

// Тепер:
// a: strong_count = 2 (змінна + b.next)
// b: strong_count = 2 (змінна + a.next)

// Коли a і b виходять зі scope:
// a: strong_count = 1 (все ще b.next)
// b: strong_count = 1 (все ще a.next)
// count ніколи не стане 0 — MEMORY LEAK!
```

Рішення: Weak<T> — слабкі посилання

```rust
use std::rc::{Rc, Weak};

// Weak не володіє даними, не збільшує strong_count
let strong = Rc::new(42);
let weak: Weak<i32> = Rc::downgrade(&strong);

println!("strong_count: {}", Rc::strong_count(&strong));  // 1
println!("weak_count: {}", Rc::weak_count(&strong));      // 1

// Weak::upgrade — спроба отримати Rc
match weak.upgrade() {
    Some(rc) => println!("Value: {}", rc),  // Value: 42
    None => println!("Data was dropped"),
}

// Якщо strong dropped — weak.upgrade() поверне None
drop(strong);

match weak.upgrade() {
    Some(rc) => println!("Value: {}", rc),
    None => println!("Data was dropped"),  // Цей варіант!
}
```

```rust
use std::rc::{Rc, Weak};
use std::cell::RefCell;

// Дерево: батько має strong ref на дітей, діти — weak ref на батька
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,        // Weak — не створює цикл!
    children: RefCell<Vec<Rc<Node>>>,   // Strong — володіємо дітьми
}

impl Node {
    fn new(value: i32) -> Rc<Node> {
        Rc::new(Node {
            value,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(Vec::new()),
        })
    }

    fn add_child(parent: &Rc<Node>, child: Rc<Node>) {
        *child.parent.borrow_mut() = Rc::downgrade(parent);
        parent.children.borrow_mut().push(child);
    }
}

// Тепер батько володіє дітьми, але діти не володіють батьком
// Коли батько dropped — діти теж dropped
// Weak посилання на батька стають invalid
```

```rust
use std::cell::RefCell;

let data = RefCell::new(5);

// borrow() — immutable borrow (runtime check)
let r1 = data.borrow();
println!("{}", r1);  // 5

// borrow_mut() — mutable borrow (runtime check)
drop(r1);  // Звільняємо immutable borrow

let mut r2 = data.borrow_mut();
*r2 += 10;
println!("{}", r2);  // 15

// ⚠️ Порушення правил = PANIC у runtime!
// let r3 = data.borrow();     // immutable
// let r4 = data.borrow_mut(); // ❌ panic! вже є immutable
```

```
// try_borrow() / try_borrow_mut() — без panic
match data.try_borrow_mut() {
    Ok(mut r) => *r += 1,
    Err(_) => println!("Already borrowed!"),
}
```

```
let x = RefCell::new(5);
let r1 = x.borrow();     // immutable
```

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Rc<RefCell<T>> — множинне володіння + мутабельність
let shared_data = Rc::new(RefCell::new(vec![1, 2, 3]));

let owner1 = Rc::clone(&shared_data);
let owner2 = Rc::clone(&shared_data);

// owner1 змінює дані
owner1.borrow_mut().push(4);

// owner2 бачить зміни!
println!("{:?}", owner2.borrow());  // [1, 2, 3, 4]

// Обидва можуть читати
println!("owner1: {:?}", owner1.borrow());
println!("owner2: {:?}", owner2.borrow());

// Типовий патерн у Rust для спільного мутабельного стану
type SharedState<T> = Rc<RefCell<T>>;
```

⚠️ Rc<RefCell<T>> — лише для однопоточних! Для потоків: Arc<Mutex<T>>

```rust
use std::cell::Cell;

// Cell<T> — interior mutability для Copy типів
// Простіше за RefCell, без borrow tracking

let data = Cell::new(5);

// get() — копіює значення (T: Copy)
let value = data.get();  // 5

// set() — замінює значення
data.set(10);
println!("{}", data.get());  // 10

// Корисно для лічильників, прапорців
struct Counter {
    count: Cell<u32>,
}

impl Counter {
    fn increment(&self) {  // &self, не &mut self!
        self.count.set(self.count.get() + 1);
    }
}

let counter = Counter { count: Cell::new(0) };
counter.increment();
counter.increment();
println!("Count: {}", counter.count.get());  // 2
```

```
// Коли що використовувати:
// Cell — для простих Copy типів (i32, bool, f64)
// RefCell — для складних типів (String, Vec, structs)

struct Config {
    debug_mode: Cell<bool>,           // Простий прапорець
    log_buffer: RefCell<Vec<String>>, // Складний тип
}
```

```rust
use std::rc::Rc;
use std::cell::RefCell;
use std::collections::HashMap;

type SharedMap = Rc<RefCell<HashMap<Position, CellType>>>;

struct WorldMap {
    cells: SharedMap,
}

impl WorldMap {
    fn new() -> Self {
        WorldMap {
            cells: Rc::new(RefCell::new(HashMap::new())),
        }
    }

    fn share(&self) -> SharedMap {
        Rc::clone(&self.cells)
    }
}

// Кілька агентів мають доступ до карти
let map = WorldMap::new();
let drone1_map = map.share();
let drone2_map = map.share();

// Drone1 оновлює
drone1_map.borrow_mut().insert(Position::new(10, 20), CellType::Obstacle);
```

```rust
use std::rc::{Rc, Weak};
use std::cell::RefCell;

struct Agent {
    id: u32,
    name: String,
    // Сусіди — Weak щоб уникнути циклів
    neighbors: RefCell<Vec<Weak<Agent>>>,
}

impl Agent {
    fn new(id: u32, name: &str) -> Rc<Agent> {
        Rc::new(Agent {
            id,
            name: name.to_string(),
            neighbors: RefCell::new(Vec::new()),
        })
    }

    fn connect(a: &Rc<Agent>, b: &Rc<Agent>) {
        a.neighbors.borrow_mut().push(Rc::downgrade(b));
        b.neighbors.borrow_mut().push(Rc::downgrade(a));
    }

    fn broadcast(&self, message: &str) {
        for weak_neighbor in self.neighbors.borrow().iter() {
            if let Some(neighbor) = weak_neighbor.upgrade() {
                println!("{} -> {}: {}", self.name, neighbor.name, message);
            }
        }
```

```rust
use std::rc::{Rc, Weak};
use std::cell::RefCell;

trait Observer {
    fn on_event(&self, event: &Event);
}

struct EventBus {
    // Weak — підписники можуть бути видалені
    observers: RefCell<Vec<Weak<dyn Observer>>>,
}

impl EventBus {
    fn new() -> Rc<Self> {
        Rc::new(EventBus {
            observers: RefCell::new(Vec::new()),
        })
    }

    fn subscribe(&self, observer: &Rc<dyn Observer>) {
        self.observers.borrow_mut().push(Rc::downgrade(observer));
    }

    fn notify(&self, event: &Event) {
        // Очищуємо мертві посилання і нотифікуємо живі
        self.observers.borrow_mut().retain(|weak| {
            if let Some(observer) = weak.upgrade() {
                observer.on_event(event);
                true  // Зберігаємо
            } else {
```

```rust
use std::rc::{Rc, Weak};
use std::cell::RefCell;
use std::collections::HashMap;

/// Кеш, що не запобігає очищенню
struct WeakCache<K, V> {
    cache: RefCell<HashMap<K, Weak<V>>>,
}

impl<K: Eq + Hash, V> WeakCache<K, V> {
    fn new() -> Self {
        WeakCache { cache: RefCell::new(HashMap::new()) }
    }

    fn insert(&self, key: K, value: &Rc<V>) {
        self.cache.borrow_mut().insert(key, Rc::downgrade(value));
    }

    fn get(&self, key: &K) -> Option<Rc<V>> {
        self.cache.borrow().get(key)?.upgrade()
    }

    /// Очистити недійсні записи
    fn cleanup(&self) {
        self.cache.borrow_mut().retain(|_, weak| weak.strong_count() > 0);
    }
}

// Кешуємо розраховані маршрути
let route_cache: WeakCache<(Position, Position), Route> = WeakCache::new();
```

```rust
use std::rc::Rc;
use std::cell::RefCell;

#[derive(Debug, Clone)]
struct SwarmConfig {
    max_speed: f64,
    communication_range: f64,
    formation: FormationType,
}

type SharedConfig = Rc<RefCell<SwarmConfig>>;

struct Drone {
    id: u32,
    config: SharedConfig,  // Спільна конфігурація
}

impl Drone {
    fn new(id: u32, config: SharedConfig) -> Self {
        Drone { id, config }
    }

    fn get_max_speed(&self) -> f64 {
        self.config.borrow().max_speed
    }
}

// Командир може змінити конфігурацію для всіх
fn update_formation(config: &SharedConfig, formation: FormationType) {
    config.borrow_mut().formation = formation;
```

```rust
use std::cell::OnceCell;

// OnceCell — ініціалізація один раз
struct Config {
    data: OnceCell<String>,
}

impl Config {
    fn new() -> Self {
        Config { data: OnceCell::new() }
    }

    fn get_or_init(&self) -> &String {
        self.data.get_or_init(|| {
            println!("Initializing...");
            expensive_computation()
        })
    }
}

let config = Config::new();
println!("{}", config.get_or_init());  // Initializing... result
println!("{}", config.get_or_init());  // result (без Initializing)

// LazyCell (Rust 1.80+) — lazy initialization
use std::cell::LazyCell;

let lazy: LazyCell<String> = LazyCell::new(|| {
    println!("Computing...");
    "result".to_string()
```

Порівняння всіх Smart Pointers

Тип	Heap	Shared	Mut	Thread	Коли
Box<T>	✓	❌	✓	✓*	Рекурсія, dyn
Rc<T>	✓	✓	❌	❌	Shared read
Rc<RefCell<T>>	✓	✓	✓ⁱ	❌	Shared mut
Arc<T>	✓	✓	❌	✓	Thread shared
Arc<Mutex<T>>	✓	✓	✓ˡ	✓	Thread mut
Cell<T>	❌	❌	✓ⁱ	❌	Copy interior
RefCell<T>	❌	❌	✓ⁱ	❌	Any interior

* якщо T: Send, ⁱ interior mutability, ˡ lock-based

Типові помилки

```
a.next = Rc::clone(&b);
b.next = Rc::clone(&a);
// Memory leak!
```

✓ Використовуйте Weak

```
a.next = Rc::clone(&b);
b.parent = Rc::downgrade(&a);
// ✓ No leak
```

Практичні поради

✓ Починайте з ownership — Box тільки коли потрібно
✓ Rc тільки для спільного володіння
✓ Weak для "батьківських" посилань у деревах/графах
✓ RefCell — останній resort для interior mutability
✓ Використовуйте try_borrow() для безпечного коду

❌ Не використовуйте Rc без потреби — overhead
❌ Не забувайте про Weak для циклічних структур
❌ Rc + RefCell у багатопотоковому — Arc + Mutex

🤖 Для MAC:
• Shared map: Rc<RefCell<HashMap<...>>>
• Agent graph: Rc<Agent> з Weak для сусідів
• Observer: Weak<dyn Observer>
• Config: Rc<RefCell<Config>>

Підсумок лекції

Box<T>:
- Heap allocation, єдиний власник
- Рекурсивні типи, dyn Trait

Rc<T>:
- Reference counting, спільне володіння
- Тільки immutable, single-thread

Weak<T>:
- Слабкі посилання, уникнення циклів
- upgrade() → Option<Rc<T>>

RefCell<T> / Cell<T>:
- Interior mutability
- Runtime borrow checking

→ Наступна лекція: Arc, Mutex, багатопотоковість

Rc<RefCell<T>>: спільне + мутабельне

Завдання для самостійної роботи

1. Linked List: Реалізуйте зв'язаний список з Box<Node>.
   Методи: push_front, pop_front, len, iter.

2. Binary Tree: Створіть бінарне дерево пошуку.
   Методи: insert, contains, height.

3. Shared Counter: Rc<RefCell<u32>> з кількома власниками.
   Демонстрація спільного стану.

4. MAC Graph: Граф агентів:
   - Rc<Agent> для вузлів
   - Weak для сусідів
   - Методи: connect, broadcast, find_path

5. Observer: Реалізуйте EventBus з Weak<dyn Observer>.
   Автоматичне очищення видалених підписників.

Дякую за увагу!

Box • Rc • Weak • RefCell • Cell

Питання?