Лекція 8

Узагальнення (Generics)

Параметричний поліморфізм у Rust

<T> • bounds • where • monomorphization

🤖 Приклади: узагальнені контейнери, алгоритми для агентів

Частина 1: Основи generics

О. С. Бичков • 2025

План лекції (Частина 1)

1. Проблема дублювання коду
2. Що таке generics?
3. Generic функції
4. Generic структури
5. Generic enum
6. Generic методи (impl)
7. Turbofish синтаксис
8. Monomorphization
9. Generics vs Trait Objects
10. Кілька параметрів типу
11. Generic константи (const generics)
12. 🤖 Generic контейнер для агентів
13. 🤖 Узагальнена черга команд
14. 🤖 Generic Result для місій
15. PhantomData

Частина 2: Trait bounds, where клаузи, просунуті патерни

```rust
// Знайти найбільше число в масиві i32
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

// Та сама логіка для f64!
fn largest_f64(list: &[f64]) -> &f64 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

// І для char... і для String... КОПІПАСТ! 😫
```

Потрібен спосіб написати функцію ОДИН раз для БУДЬ-ЯКОГО типу!

```rust
// <T> — параметр типу (type parameter)
// T — placeholder для конкретного типу
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

// Одна функція для всіх типів!
let numbers = vec![34, 50, 25, 100, 65];
let result = largest(&numbers);  // T = i32

let chars = vec!['y', 'm', 'a', 'q'];
let result = largest(&chars);    // T = char
```

```rust
// Один параметр типу
fn foo<T>(x: T) { }

// Кілька параметрів
fn bar<T, U>(x: T, y: U) { }

// 3 trait bounds
fn baz<T: Clone>(x: T) { }

// Конвенції іменування:
// T — Type (загальний тип)
// E — Error (тип помилки)
// K, V — Key, Value (для map)
// R — Return (тип повернення)
```

Параметри типу завжди в кутових дужках <T> після імені функції/структури/enum

```rust
// Проста generic функція
fn identity<T>(x: T) -> T {
    x
}

let int = identity(5);       // T = i32
let str = identity("hello"); // T = &str

// Generic функція з кількома параметрами
fn swap<T, U>(pair: (T, U)) -> (U, T) {
    (pair.1, pair.0)
}

let result = swap((1, "hello"));  // ("hello", 1)

// Generic з посиланнями
fn first<T>(slice: &[T]) -> Option<&T> {
    slice.first()
}

let nums = vec![1, 2, 3];
let first_num = first(&nums);  // Some(&1)
```

```rust
// Структура з параметром типу
struct Point<T> {
    x: T,
    y: T,
}

// Використання
let integer_point = Point { x: 5, y: 10 };      // Point<i32>
let float_point = Point { x: 1.0, y: 4.0 };     // Point<f64>

// Різні типи для різних полів
struct Pair<T, U> {
    first: T,
    second: U,
}

let pair = Pair { first: 5, second: "hello" };  // Pair<i32, &str>

// Вкладені generics
struct Container<T> {
    items: Vec<T>,  // Vec теж generic!
}

let container: Container<i32> = Container { items: vec![1, 2, 3] };
```

```
// Option<T> — стандартний generic enum
enum Option<T> {
    Some(T),
    None,
}

// Result<T, E> — два параметри типу
enum Result<T, E> {
    Ok(T),
    Err(E),
}

// Власний generic enum
enum BinaryTree<T> {
    Leaf(T),
    Node {
        value: T,
        left: Box<BinaryTree<T>>,
        right: Box<BinaryTree<T>>,
    },
    Empty,
}

let tree: BinaryTree<i32> = BinaryTree::Node {
    value: 10,
    left: Box::new(BinaryTree::Leaf(5)),
    right: Box::new(BinaryTree::Leaf(15)),
};
```

```rust
struct Point<T> {
    x: T,
    y: T,
}

// impl для ВСІХ T
impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x, y }
    }

    fn x(&self) -> &T {
        &self.x
    }
}

// impl тільки для конкретного типу
impl Point<f64> {
    fn distance_from_origin(&self) -> f64 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

let p1 = Point::new(5, 10);      // Point<i32>
let p2 = Point::new(1.0, 2.0);   // Point<f64>

// p1.distance_from_origin();  // ❌ Error — тільки для f64
p2.distance_from_origin();      // ✓ 2.236...
```

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    // Метод з власним параметром типу U
    fn mixup<U>(self, other: Point<U>) -> Point<T> {
        Point {
            x: self.x,   // T від self
            y: other.y,  // Помилка! Типи не збігаються
        }
    }
}

// Правильний варіант — повертаємо змішану структуру
impl<T, U> Point<T> {
    fn mixup<V, W>(self, other: Point<V>) -> (Point<T>, Point<V>) {
        // Повертаємо обидві точки
        (self, other)
    }
}

// Або створюємо нову структуру
struct MixedPoint<X, Y> { x: X, y: Y }

impl<T> Point<T> {
    fn mixup<U>(self, other: Point<U>) -> MixedPoint<T, U> {
        MixedPoint { x: self.x, y: other.y }
    }
```

```
// Зазвичай компілятор виводить типи автоматично
let v = vec![1, 2, 3];  // Vec<i32> — виведено

// Але іноді потрібно вказати явно
// Turbofish syntax: ::<Type>

// Приклад 1: parse()
let num: i32 = "42".parse().unwrap();     // Через анотацію типу
let num = "42".parse::<i32>().unwrap();   // Через turbofish

// Приклад 2: collect()
let chars: Vec<char> = "hello".chars().collect();  // Анотація
let chars = "hello".chars().collect::<Vec<char>>(); // Turbofish
let chars = "hello".chars().collect::<Vec<_>>();   // _ = виведи сам

// Приклад 3: Default
let default = i32::default();              // 0
let default = String::default();           // ""
let default = <Vec<i32>>::default();       // []

// Коли turbofish необхідний?
// Коли компілятор не може вивести тип з контексту
```

Monomorphization — магія zero-cost

```
fn id<T>(x: T) -> T { x }

id(5);     // i32
id(3.14);  // f64
id("hi");  // &str
```

Після monomorphization

Zero-cost: generic код працює так само швидко як написаний вручну!
Недолік: більший бінарний розмір (code bloat)

```
fn id_i32(x: i32) -> i32 { x }
fn id_f64(x: f64) -> f64 { x }
fn id_str(x: &str) -> &str { x }
```

Generics vs Trait Objects

```
fn process<T: Agent>(agent: &T) {
    agent.tick();
}
```

- ✓ Zero-cost (monomorphization)
- ✓ Інлайн можливий
- ✓ Оптимізації компілятора
- ✗ Більший бінарний файл
- ✗ Тип відомий на компіляції
- ✗ Не гетерогенні колекції

Trait Objects (динамічний dispatch)

- ✓ Один екземпляр коду
- ✓ Гетерогенні колекції
- ✓ Тип в runtime
- ✗ Vtable lookup overhead
- ✗ Немає інлайну
- ✗ Object safety обмеження

Правило: generics за замовчуванням, dyn коли потрібна runtime гнучкість

```
fn process(agent: &dyn Agent) {
    agent.tick();
}
```

```rust
// Два параметри типу
struct KeyValue<K, V> {
    key: K,
    value: V,
}

let item: KeyValue<String, i32> = KeyValue {
    key: String::from("age"),
    value: 30,
};

// Функція з кількома параметрами
fn combine<A, B, C>(a: A, b: B, f: impl Fn(A, B) -> C) -> C {
    f(a, b)
}

let result = combine(5, 3, |a, b| a + b);  // 8

// HashMap — класичний приклад
use std::collections::HashMap;
let mut map: HashMap<String, Vec<i32>> = HashMap::new();
map.insert("numbers".to_string(), vec![1, 2, 3]);

// Result з двома параметрами
fn parse_config() -> Result<Config, ConfigError> {
    // ...
}
```

```rust
// Const generic — параметр-значення, а не тип
// Rust 1.51+

// Масив з параметризованим розміром
struct ArrayWrapper<T, const N: usize> {
    data: [T; N],
}

impl<T: Default + Copy, const N: usize> ArrayWrapper<T, N> {
    fn new() -> Self {
        ArrayWrapper {
            data: [T::default(); N],
        }
    }
}

let arr3: ArrayWrapper<i32, 3> = ArrayWrapper::new();  // [0, 0, 0]
let arr5: ArrayWrapper<i32, 5> = ArrayWrapper::new();  // [0, 0, 0, 0, 0]

// Практичний приклад: буфер фіксованого розміру
struct Buffer<const SIZE: usize> {
    data: [u8; SIZE],
    len: usize,
}

let small_buffer: Buffer<64> = Buffer { data: [0; 64], len: 0 };
let large_buffer: Buffer<1024> = Buffer { data: [0; 1024], len: 0 };
```

```
/// Узагальнений реєстр сутностей
struct Registry<T> {
    items: HashMap<u32, T>,
    next_id: u32,
}

impl<T> Registry<T> {
    fn new() -> Self {
        Registry {
            items: HashMap::new(),
            next_id: 1,
        }
    }

    fn register(&mut self, item: T) -> u32 {
        let id = self.next_id;
        self.items.insert(id, item);
        self.next_id += 1;
        id
    }

    fn get(&self, id: u32) -> Option<&T> {
        self.items.get(&id)
    }

    fn remove(&mut self, id: u32) -> Option<T> {
        self.items.remove(&id)
    }

    fn count(&self) -> usize {
```

```rust
/// Generic черга з пріоритетами
struct PriorityQueue<T> {
    items: Vec<(u8, T)>,  // (priority, item)
}

impl<T> PriorityQueue<T> {
    fn new() -> Self {
        PriorityQueue { items: Vec::new() }
    }

    fn push(&mut self, priority: u8, item: T) {
        self.items.push((priority, item));
        // Сортуємо за пріоритетом (вищий — перший)
        self.items.sort_by(|a, b| b.0.cmp(&a.0));
    }

    fn pop(&mut self) -> Option<T> {
        if self.items.is_empty() {
            None
        } else {
            Some(self.items.remove(0).1)
        }
    }

    fn is_empty(&self) -> bool {
        self.items.is_empty()
    }
}

// Для команд, місій, повідомлень...
```

```rust
/// Результат операції агента
enum AgentResult<T, E> {
    Success(T),
    Failure(E),
    Pending,      // Ще виконується
    Cancelled,    // Скасовано
}

impl<T, E> AgentResult<T, E> {
    fn is_success(&self) -> bool {
        matches!(self, AgentResult::Success(_))
    }

    fn unwrap(self) -> T {
        match self {
            AgentResult::Success(v) => v,
            _ => panic!("Called unwrap on non-success"),
        }
    }

    fn map<U, F: FnOnce(T) -> U>(self, f: F) -> AgentResult<U, E> {
        match self {
            AgentResult::Success(v) => AgentResult::Success(f(v)),
            AgentResult::Failure(e) => AgentResult::Failure(e),
            AgentResult::Pending => AgentResult::Pending,
            AgentResult::Cancelled => AgentResult::Cancelled,
        }
    }
}
```

```
/// Узагальнена подія
struct Event<T> {
    timestamp: u64,
    source_id: u32,
    payload: T,
}

impl<T> Event<T> {
    fn new(source_id: u32, payload: T) -> Self {
        Event {
            timestamp: current_time(),
            source_id,
            payload,
        }
    }
}

// Різні типи подій
struct PositionUpdate { x: f64, y: f64, altitude: f64 }
struct BatteryStatus { level: u8, charging: bool }
struct TargetDetected { target_id: u32, confidence: f32 }

// Типізовані події
type PositionEvent = Event<PositionUpdate>;
type BatteryEvent = Event<BatteryStatus>;
type DetectionEvent = Event<TargetDetected>;

// Обробник подій
fn handle_event<T: Debug>(event: Event<T>) {
    println!("[{}] Agent {}: {:?}", event.timestamp, event.source_id, event.payload);
```

```rust
use std::marker::PhantomData;

// Проблема: T не використовується в полях
struct Id<T> {
    value: u64,
    // ❌ Error: T не використовується
}

// Рішення: PhantomData
struct Id<T> {
    value: u64,
    _marker: PhantomData<T>,  // "Фантомне" поле
}

// Типобезпечні ID
struct Drone;
struct Mission;

type DroneId = Id<Drone>;
type MissionId = Id<Mission>;

impl<T> Id<T> {
    fn new(value: u64) -> Self {
        Id { value, _marker: PhantomData }
    }
}

let drone_id: DroneId = Id::new(42);
let mission_id: MissionId = Id::new(42);
```

```rust
use std::marker::PhantomData;

// Маркери одиниць
struct Meters;
struct Kilometers;
struct Feet;

// Типобезпечна відстань
struct Distance<Unit> {
    value: f64,
    _unit: PhantomData<Unit>,
}

impl<Unit> Distance<Unit> {
    fn new(value: f64) -> Self {
        Distance { value, _unit: PhantomData }
    }
}

// Конвертація
impl Distance<Meters> {
    fn to_kilometers(self) -> Distance<Kilometers> {
        Distance::new(self.value / 1000.0)
    }
}

let altitude: Distance<Meters> = Distance::new(1500.0);
let altitude_km: Distance<Kilometers> = altitude.to_kilometers();

// ❌ Компілятор не дасть змішати
```

```rust
#[derive(Debug)]
struct Config<T> {
    value: T,
    enabled: bool,
    retries: u32,
}

// Default тільки коли T: Default
impl<T: Default> Default for Config<T> {
    fn default() -> Self {
        Config {
            value: T::default(),
            enabled: true,
            retries: 3,
        }
    }
}

// Працює для типів з Default
let int_config: Config<i32> = Config::default();
// Config { value: 0, enabled: true, retries: 3 }

let string_config: Config<String> = Config::default();
// Config { value: "", enabled: true, retries: 3 }

// Частковий override
let custom = Config {
    value: 100,
    ..Default::default()
};
```

```
// Generics всередині generics
struct Response<T> {
    data: Option<T>,        // Option<T>
    errors: Vec<String>,    // Vec<String>
}

// Result з generic типами
type ApiResult<T> = Result<Response<T>, ApiError>;

// HashMap з generic value
struct Cache<K, V> {
    store: HashMap<K, Option<V>>,
    ttl: Duration,
}

// Складні вкладення
struct MultiAgentSystem<A, M> {
    agents: HashMap<u32, A>,
    mailboxes: HashMap<u32, Vec<M>>,
    pending: Vec<(u32, M)>,
}

// Використання
let mut mas: MultiAgentSystem<Drone, DroneMessage> = MultiAgentSystem {
    agents: HashMap::new(),
    mailboxes: HashMap::new(),
    pending: Vec::new(),
};
```

Синтаксис generics — зведення

Конструкція	Синтаксис	Приклад
Функція	fn name<T>(...)	fn id<T>(x: T) -> T
Структура	struct Name<T> {...}	struct Point<T> { x: T }
Enum	enum Name<T> {...}	enum Option<T> { Some(T) }
Impl	impl<T> Name<T> {...}	impl<T> Point<T> { ... }
Trait bound	<T: Trait>	<T: Clone + Debug>
Const generic	<const N: usize>	struct Arr<const N: usize>
Turbofish	name::<Type>(...)	parse::<i32>()

Підсумок: Частина 1

Generics — параметризація типами:
- <T> — параметр типу
- Один код для багатьох типів
- Zero-cost через monomorphization

Где використовувати:
- Функції: fn name<T>(...)
- Структури: struct Name<T> {...}
- Enum: enum Name<T> {...}
- Методи: impl<T> Name<T> {...}

Додатково:
- Const generics для значень
- PhantomData для маркерів типу
- Turbofish для явного типу

→ Частина 2: Trait bounds, where клаузи, просунуті патерни

Лекція 8 (продовження)

Trait Bounds та Where клаузи

Обмеження на параметри типу

bounds • where • Sized • lifetimes

🤖 Приклади: обмеження для агентів, серіалізація, порівняння

Частина 2: Trait bounds та просунуті патерни

О. С. Бичков • 2025

План лекції (Частина 2)

1. Навіщо потрібні bounds?
2. Синтаксис trait bounds
3. Множинні bounds (+)
4. where клаузи
5. Bounds на асоційовані типи
6. impl Trait в аргументах
7. impl Trait у поверненні
8. Sized trait
9. ?Sized — unsized types
10. Lifetime bounds
11. Higher-Ranked Trait Bounds
12. 🤖 Bounds для агентів
13. 🤖 Серіалізація стану
14. 🤖 Comparable агенти
15. 🤖 Generic координатор
16. Типові помилки

```rust
// Без bounds — T може бути БУДЬ-ЯКИМ типом
fn print_value<T>(value: T) {
    println!("{}", value);  // ❌ Error!
    // T не обов'язково реалізує Display!
}

// З bound — гарантуємо що T реалізує Display
fn print_value<T: Display>(value: T) {
    println!("{}", value);  // ✓ Працює!
}

// Ще приклад
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {  // ❌ Error! T не обов'язково порівнюваний
            largest = item;
        }
    }
    largest
}

// З bound
fn largest<T: PartialOrd>(list: &[T]) -> &T { ... }  // ✓
```

Bounds обмежують T до типів з потрібною функціональністю

```rust
// Синтаксис 1: Після параметра типу
fn function<T: TraitName>(arg: T) { }

// Синтаксис 2: impl Trait (скорочення)
fn function(arg: impl TraitName) { }

// Обидва еквівалентні!

// Приклади
fn debug_print<T: Debug>(item: &T) {
    println!("{:?}", item);
}

fn clone_and_print<T: Clone + Display>(item: &T) {
    let copy = item.clone();
    println!("{}", copy);
}

// Структури з bounds
struct Wrapper<T: Display> {
    value: T,
}

// Методи з bounds
impl<T: Clone> MyStruct<T> {
    fn duplicate(&self) -> Self { ... }
}
```

```rust
// + для комбінації кількох traits
fn process<T: Clone + Debug + Display>(item: T) {
    let copy = item.clone();       // Clone
    println!("{:?}", copy);        // Debug
    println!("{}", copy);          // Display
}

// Різні bounds для різних параметрів
fn compare_and_display<T: PartialOrd, U: Display>(a: T, b: T, label: U) {
    if a < b {
        println!("{}: a < b", label);
    }
}

// Приклад з багатьма traits
fn serialize_and_hash<T>(item: &T) -> u64
where
    T: Serialize + Hash + Debug,
{
    println!("Serializing: {:?}", item);
    let json = serde_json::to_string(item).unwrap();
    calculate_hash(item)
}
```

```rust
// Проблема: bounds у сигнатурі стають нечитабельними
fn some_function<T: Display + Clone, U: Clone + Debug, V: PartialOrd + Hash>(
    t: T, u: U, v: V
) -> i32 {
    // ...
}

// Рішення: where клауза
fn some_function<T, U, V>(t: T, u: U, v: V) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
    V: PartialOrd + Hash,
{
    // ...
}

// where дозволяє складніші bounds
fn process<T, F>(items: T, f: F)
where
    T: IntoIterator,           // T — ітерабельний
    T::Item: Display,          // Елементи T реалізують Display
    F: Fn(&T::Item) -> bool,   // F — функція-предикат
{
    for item in items {
        if f(&item) {
            println!("{}", item);
        }
    }
}
```

```rust
// Асоційований тип з bound
trait Container {
    type Item;

    fn get(&self, index: usize) -> Option<&Self::Item>;
}

// Функція з bound на асоційований тип
fn print_first<C>(container: &C)
where
    C: Container,
    C::Item: Display,  // Bound на асоційований тип!
{
    if let Some(item) = container.get(0) {
        println!("First: {}", item);
    }
}

// Iterator з bound на Item
fn sum_positive<I>(iter: I) -> i32
where
    I: Iterator<Item = i32>,  // Item = i32
{
    iter.filter(|x| *x > 0).sum()
}

// Або коротше
fn sum_positive(iter: impl Iterator<Item = i32>) -> i32 {
    iter.filter(|x| *x > 0).sum()
}
```

```rust
// impl Trait — синтаксичний цукор для простих bounds

// Ці дві функції еквівалентні:
fn print_debug<T: Debug>(item: &T) {
    println!("{:?}", item);
}

fn print_debug(item: &impl Debug) {
    println!("{:?}", item);
}

// impl Trait зручний для closures
fn apply<F: Fn(i32) -> i32>(f: F, x: i32) -> i32 {
    f(x)
}

fn apply(f: impl Fn(i32) -> i32, x: i32) -> i32 {
    f(x)
}

// Коли використовувати що?
// impl Trait — простіший синтаксис для одного параметра
// <T: Trait> — коли T використовується в кількох місцях

fn compare<T: PartialOrd>(a: &T, b: &T) -> bool {
    a < b  // T в кількох місцях
}
```

```rust
// impl Trait у return — приховує конкретний тип

// Повертаємо "щось що реалізує Iterator"
fn even_numbers() -> impl Iterator<Item = i32> {
    (0..).filter(|x| x % 2 == 0)
}

// Без impl Trait тип був би жахливим:
// -> std::iter::Filter<std::ops::RangeFrom<i32>, fn(&i32) -> bool>

// Для closures (унікальний тип)
fn make_adder(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n
}

let add_5 = make_adder(5);
println!("{}", add_5(10));  // 15

// ⚠️ Обмеження: завжди один конкретний тип!
fn get_animal(is_dog: bool) -> impl Animal {
    if is_dog {
        Dog  // ✓
    } else {
        Cat  // ❌ Error! Різні типи
    }
}
```

```
// Sized — marker trait для типів з відомим розміром на компіляції
// НЕЯВНО додається до всіх generic параметрів!

// Це:
fn foo<T>(x: T) { }

// Насправді означає:
fn foo<T: Sized>(x: T) { }

// Типи з відомим розміром:
// i32, f64, bool, char, [T; N], structs, enums...

// Типи БЕЗ відомого розміру (DST - Dynamically Sized Types):
// str — рядковий slice (не &str!)
// [T] — slice (не &[T]!)
// dyn Trait — trait object

// DST можна використовувати тільки через посилання або Box:
fn takes_str(s: &str) { }           // ✓ &str має фіксований розмір
fn takes_slice(s: &[i32]) { }       // ✓ &[T] має фіксований розмір
fn takes_dyn(a: &dyn Animal) { }    // ✓ &dyn має фіксований розмір
fn takes_box(a: Box<dyn Animal>) {} // ✓ Box має фіксований розмір
```

```
// ?Sized — "можливо не Sized", знімає неявний Sized bound

// За замовчуванням T: Sized
fn foo<T>(x: &T) { }  // T повинен бути Sized

// ?Sized дозволяє DST
fn foo<T: ?Sized>(x: &T) { }  // T може бути unsized

// Практичний приклад
fn print_it<T: Display>(item: &T) {
    println!("{}", item);
}

print_it(&"hello");  // ✓ &str
// print_it::<str>(&"hello");  // ❌ str не Sized

// З ?Sized
fn print_it<T: Display + ?Sized>(item: &T) {
    println!("{}", item);
}

print_it::<str>("hello");  // ✓ Тепер працює!

// Стандартна бібліотека використовує ?Sized:
impl<T: ?Sized> AsRef<T> for T { ... }
impl<T: ?Sized> Borrow<T> for T { ... }
```

```rust
// Lifetime як bound — T має жити принаймні 'a
fn longest<'a, T>(x: &'a T, y: &'a T) -> &'a T
where
    T: PartialOrd,
{
    if x > y { x } else { y }
}

// T: 'a — T може містити посилання, але вони мають жити 'a
struct Wrapper<'a, T: 'a> {
    reference: &'a T,
}

// T: 'static — T не містить non-static посилань
fn spawn_thread<T: Send + 'static>(data: T) {
    std::thread::spawn(move || {
        // data використовується в новому потоці
    });
}

// Lifetime + trait bounds
fn process<'a, T>(item: &'a T)
where
    T: Display + 'a,
{
    println!("{}", item);
}
```

```rust
// HRTB — для функцій що працюють з будь-яким lifetime

// Проблема: closure що приймає посилання
fn apply_to_ref<F>(f: F)
where
    F: Fn(&i32) -> i32,  // Який lifetime у &i32?
{
    let x = 10;
    let result = f(&x);
}

// HRTB: for<'a> — "для будь-якого lifetime 'a"
fn apply_to_ref<F>(f: F)
where
    F: for<'a> Fn(&'a i32) -> i32,
{
    let x = 10;
    let result = f(&x);
}

// На практиці Rust виводить це автоматично для Fn traits:
fn apply_to_ref(f: impl Fn(&i32) -> i32) {
    // Компілятор додає for<'a> неявно
}

apply_to_ref(|x| x + 1);  // ✓
```

HRTB рідко потрібні явно — компілятор зазвичай виводить їх сам

```rust
use std::fmt::Debug;
use serde::{Serialize, Deserialize};

/// Вимоги до стану агента
trait AgentState: Clone + Debug + Default {}

/// Вимоги до повідомлень
trait Message: Clone + Debug + Send + Sync {}

/// Generic агент з bounded типами
struct Agent<S, M>
where
    S: AgentState,
    M: Message,
{
    id: u32,
    state: S,
    inbox: Vec<M>,
    outbox: Vec<(u32, M)>,
}

impl<S, M> Agent<S, M>
where
    S: AgentState,
    M: Message,
{
    fn new(id: u32) -> Self {
        Agent {
            id,
            state: S::default(),  // Можливо завдяки Default bound
```

```rust
use serde::{Serialize, Deserialize, de::DeserializeOwned};

/// Персистентний агент — можна зберігати/відновлювати
trait Persistent {
    fn save(&self) -> Result<Vec<u8>, SaveError>;
    fn load(data: &[u8]) -> Result<Self, LoadError>
    where
        Self: Sized;
}

// Generic реалізація для всіх Serialize + DeserializeOwned
impl<T> Persistent for T
where
    T: Serialize + DeserializeOwned,
{
    fn save(&self) -> Result<Vec<u8>, SaveError> {
        bincode::serialize(self).map_err(SaveError::from)
    }

    fn load(data: &[u8]) -> Result<Self, LoadError> {
        bincode::deserialize(data).map_err(LoadError::from)
    }
}

// Будь-який Serialize + Deserialize автоматично Persistent!
#[derive(Serialize, Deserialize)]
struct DroneState { position: Position, battery: u8 }

let state = DroneState { /* ... */ };
let bytes = state.save()?;  // ✓ Працює автоматично!
```

```rust
use std::cmp::Ordering;

/// Агент що можна порівнювати за пріоритетом
trait Prioritized {
    fn priority(&self) -> u8;
}

/// Пріоритетна черга для будь-яких Prioritized
struct PriorityQueue<T: Prioritized> {
    items: Vec<T>,
}

impl<T: Prioritized> PriorityQueue<T> {
    fn new() -> Self {
        PriorityQueue { items: Vec::new() }
    }

    fn push(&mut self, item: T) {
        self.items.push(item);
        self.items.sort_by(|a, b| b.priority().cmp(&a.priority()));
    }

    fn pop_highest(&mut self) -> Option<T> {
        if self.items.is_empty() {
            None
        } else {
            Some(self.items.remove(0))
        }
    }
}
```

```rust
/// Trait для типів, що можуть бути агентами
trait AgentLike: Debug + Send {
    fn id(&self) -> u32;
    fn tick(&mut self);
    fn is_active(&self) -> bool;
}

/// Generic координатор
struct Coordinator<A>
where
    A: AgentLike,
{
    agents: HashMap<u32, A>,
    tick_count: u64,
}

impl<A: AgentLike> Coordinator<A> {
    fn new() -> Self {
        Coordinator {
            agents: HashMap::new(),
            tick_count: 0,
        }
    }

    fn register(&mut self, agent: A) {
        let id = agent.id();
        self.agents.insert(id, agent);
    }

    fn tick_all(&mut self) {
```

```rust
/// Trait для створення агентів
trait AgentFactory {
    type Agent: AgentLike;
    type Config;

    fn create(&self, id: u32, config: Self::Config) -> Self::Agent;
}

/// Фабрика дронів
struct DroneFactory;

impl AgentFactory for DroneFactory {
    type Agent = Drone;
    type Config = DroneConfig;

    fn create(&self, id: u32, config: DroneConfig) -> Drone {
        Drone::new(id, config)
    }
}

/// Generic spawner
fn spawn_agents<F, C>(factory: &F, configs: Vec<C>) -> Vec<F::Agent>
where
    F: AgentFactory<Config = C>,
{
    configs
        .into_iter()
        .enumerate()
        .map(|(i, config)| factory.create(i as u32, config))
        .collect()
```

```rust
/// Trait для обробки повідомлень
trait MessageHandler<M> {
    type Response;

    fn handle(&mut self, message: M) -> Self::Response;
}

/// Дрон обробляє різні типи повідомлень
impl MessageHandler<MoveCommand> for Drone {
    type Response = Result<Position, MoveError>;

    fn handle(&mut self, cmd: MoveCommand) -> Self::Response {
        self.move_to(cmd.destination)
    }
}

impl MessageHandler<ScanCommand> for Drone {
    type Response = Result<ScanResult, SensorError>;

    fn handle(&mut self, cmd: ScanCommand) -> Self::Response {
        self.scan_area(cmd.area)
    }
}

/// Generic dispatch
fn dispatch<A, M>(agent: &mut A, message: M) -> A::Response
where
    A: MessageHandler<M>,
{
    agent.handle(message)
```

```
// Blanket impl — реалізація для всіх типів з певним bound

// Приклад зі стандартної бібліотеки
// ToString реалізований для ВСІХ Display типів
impl<T: Display> ToString for T {
    fn to_string(&self) -> String {
        format!("{}", self)
    }
}

// Власний приклад
trait Describable {
    fn describe(&self) -> String;
}

// Blanket impl для всіх Debug
impl<T: Debug> Describable for T {
    fn describe(&self) -> String {
        format!("Value: {:?}", self)
    }
}

// Тепер ВСЕ з Debug автоматично має describe()!
let num = 42;
println!("{}", num.describe());  // "Value: 42"

let vec = vec![1, 2, 3];
println!("{}", vec.describe());  // "Value: [1, 2, 3]"
```

Типові помилки з generics

```
impl<T> Display for Vec<T> {}
// ❌ Чужий trait для чужого типу
```

❌ Занадто складні bounds

Читайте повідомлення компілятора — він підкаже яких bounds не вистачає!

```
fn f<T: A+B+C+D+E+F>(x: T)
// Використайте where!
```

Коли що використовувати?

Ситуація	Рішення	Приклад
Простий bound	<T: Trait>	fn f<T: Clone>(x: T)
Аргумент функції	impl Trait	fn f(x: impl Debug)
Кілька параметрів	where	where T: A, U: B
Повернення складного типу	impl Trait return	-> impl Iterator
Гетерогенна колекція	dyn Trait	Vec<Box<dyn Agent>>
DST параметри	?Sized	<T: ?Sized>
Thread-safe	Send + Sync	<T: Send + Sync>

Практичні поради

✓ Починайте без bounds — додавайте коли компілятор вимагає
✓ Використовуйте where для читабельності
✓ impl Trait для простих випадків
✓ Документуйте чому саме такі bounds

❌ Не додавайте зайві bounds "про запас"
❌ Не ускладнюйте без потреби
❌ Уникайте глибоко вкладених generics

🤖 Для MAC:
• Clone + Debug — мінімум для стану
• Send + Sync — для багатопотоковості
• Serialize — для персистентності
• PartialOrd — для пріоритетних черг
• Default — для ініціалізації

Підсумок лекції

Generics — параметризація типами:
- <T> для функцій, структур, enum
- Zero-cost через monomorphization

Trait Bounds:
- <T: Trait> обмежує можливі типи
- + для кількох traits
- where для читабельності

Спеціальні bounds:
- Sized — розмір відомий (неявний)
- ?Sized — дозволяє DST
- 'static — для потоків

🤖 MAC: generic контейнери, координатори, фабрики
→ Наступна лекція: Lifetimes — час життя посилань

Завдання для самостійної роботи

1. Базове: Створіть generic Stack<T> з push/pop/peek.
   Додайте bound T: Clone для методу duplicate_top().

2. Bounds: Реалізуйте generic find_max<T>(slice: &[T]) -> Option<&T>
   з правильними bounds для порівняння.

3. Registry: Створіть generic Registry<K, V> де:
   - K: Hash + Eq (для HashMap)
   - V: Clone + Debug
   - Методи: register, get, remove, list_all

4. MAC: Реалізуйте generic Agent<S, M> де:
   - S: AgentState (Clone + Debug + Default)
   - M: Message (Clone + Send)
   - Методи: new, process_message, get_state

Дякую за увагу!

Generics • Trait Bounds • Where • Sized

Питання?