

# Системне Програмування

З використанням мови програмування **Rust.**  
**Fundamentals. Collections**

# Колекції

Мова програмування Rust надає потужний набір стандартних колекцій, які дозволяють ефективно працювати з даними в пам'яті. Колекції представлені в модулі `std::collections` і охоплюють різні структури даних, кожна з яких має свої особливості та застосування.



# Масив

```
let xs : [i32; 3] = [11, 13, 15];  
// length  
println!("len = {}", xs.len()); // 3  
// iterator  
for x : i32 in xs {  
    println!("x = {}", x);  
}
```

- + швидкий доступ до елементів  **$O(1)$**
- додавання елементів  **$O(N)$**
- видалення елементів  **$O(N)$**

# String

```
let mut text: String = String::from(s: "Hello");  
println!("{}", text); // Hello  
text.push(ch: '!');  
println!("{}", text); // Hello!
```

+ можемо додавати елементи (mut)  **$O(1)$  /  $O(N)$**

```
text.remove(idx: 3);  
println!("{}", text); // Helo!
```

+ можемо видаляти елементи (mut)  **$O(N)$**



# Чому при додаванні різна складність?

```
let mut text: String = String::with_capacity(capacity: 3);  
println!("{}", text); // ""  
  
text.push_str(string: "He"); // 0(2)  
println!("{}", text); // "He"  
  
text.push_str(string: "l"); // 0(1)  
println!("{}", text); // "Hel"  
  
text.push_str(string: "l"); // 0(4)  
println!("{}", text); // "Hell"
```

Тому що пам'ять що адресується, має бути розташована послідовно, і нема гарантії, що там де ми раніше виділяли пам'ять є вільна область

# Вектор

Технічно, **вектор** це дженералізований **String** оскільки може працювати з будь яким типом, не тільки **char**.

```
let mut ns: Vec<i32> = Vec::with_capacity(capacity: 3);
ns.push(value: 1); // 0(1)
println!("{:?}", ns); // [1]

ns.push(value: 2); // 0(1)
println!("{:?}", ns); // [1, 2]

ns.push(value: 3); // 0(1)
println!("{:?}", ns); // [1, 2, 3]

ns.push(value: 4); // 0(4)
println!("{:?}", ns); // [1, 2, 3, 4]
```



+ швидкий доступ до елементів  **$O(1)$**

```
let x:i32 = ns[3]; //  $O(1)$ 
```

+ швидка модифікація елементів  **$O(1)$**

```
ns[0] = 10; //  $O(1)$   
println!("{:?}", ns); // [10, 2, 3, 4]
```

— додавання елементів  **$O(1)$  /  $O(N)$**

Та сама історія що і з String

Якщо пам'ять є —  **$O(1)$**

Якщо потрібна нова алокація —  **$O(N)$**

— видалення елементів  **$O(N)$**

нова структура завжди копіюється

# Черга. VecDeque<A>

Семантично, ми можемо мати на увазі той самий масив, але структура реалізована таким чином, що

- + Доступ до елементів —  $O(1)$
- + Додавання до голови —  $O(1)$
- + Додавання до хвоста —  $O(1)$
- + Видалення з голови —  $O(1)$
- + Видалення з хвоста —  $O(1)$



# VecDeque. Приклади

```
let mut q: VecDeque<i32> = VecDeque::new();

q.push_front(value: 1); // O(1)
println!("{:?}", q); // [1]

q.push_front(value: 2); // O(1)
println!("{:?}", q); // [2, 1]

q.push_back(value: 10); // O(1)
println!("{:?}", q); // [2, 1, 10]

q.push_back(value: 20); // O(1)
println!("{:?}", q); // [2, 1, 10, 20]

let x: Option<i32> = q.pop_front(); // O(1)
println!("{:?}", q); // [1, 10, 20]

let x: Option<i32> = q.pop_back(); // O(1)
println!("{:?}", q); // [1, 10]
```

# VecDeque. Приклади

Доступ до елементів —  $O(1)$

```
let x: i32 = q[10]; //  $O(1)$ 
```

Може здатися, що це якась магія, але всьому є своя ціна

```
q.insert(index: 1, value: 33); //  $O(N)$   
println!("{:?}", q);           // [1, 33, 10]  
q.remove(index: 1);             //  $O(N)$   
println!("{:?}", q);           // [1, 10]
```

Також не забуваємо, що VecDeque знаходиться в іншому пакеті:

```
use std::collections::VecDeque;
```



# LinkedList

LinkedList — структура даних дуже схожа на VecDeque, але з деякими відмінностями:

	VecDeque	LinkedList
Додавання до голови	$O(1)$	$O(1)$
Додавання до хвоста	$O(1)$	$O(1)$
Видалення з голови	$O(1)$	$O(1)$
Видалення з хвоста	$O(1)$	$O(1)$
Доступ до елементів по індексу	$O(1)$	<b><math>O(n)</math></b>
Вставка елементів по індексу	<b><math>O(n)</math></b>	$O(1)$
Видалення елементів по індексу	<b><math>O(n)</math></b>	$O(1)$

# LinkedList. Приклади

```
// https://doc.rust-lang.org/std/collections/struct.LinkedList.html  
let mut ll : LinkedList<char> = LinkedList::new();  
ll.push_back(elt: 'a');  
ll.push_back(elt: 'b');  
ll.push_front(elt: 'x');  
ll.push_front(elt: 'y');  
println!("{:?}", ll);
```



Також LinkedList ніколи не копіюється, тому що нема такої ситуації, коли “закінчилось місце”, як то буває в **String / Vec / VecDeque**  
Чому?

Тому що всі елементи LinkedList це окремі структури даних, які “знають” тільки про попередній елемент та наступний елемент.

**Але усьому є ціна:**

- LinkedList займає пам'яті приблизно в 2 рази більше ніж Vector
- складність доступу до елемента по індексу —  $O(N)$ .

**LinkedList** — складна структура, все ще знаходиться в стадії активної розробки

# Non-Linear Data Structures

Всі наступні структури знаходяться в іншому пакеті.

```
use std::collections::*;
```



Все структури даних,  
які ми розбирали до цього моменту були **лінійні**.

Тобто ми завжди знали індекс  
і намагалися використувати доступ по індексу.

Але далеко не завжди нам це потрібно.  
Інколи нам потрібно наприклад видалити дублікати,  
або просто перевірити, чи існує такий елемент, чи ні.  
В цьому випадку індекс втрачає сенс.

Оскільки індекс втрачає сенс, то ці структури можливо зберігати  
більш оптимально і пришвидшувати деякі операції.

# Множина. HashSet<A>

Це набір унікальних значень без дублікатів.

```
let mut pp: HashSet<&str> = HashSet::new();
pp.insert(value: "Alex");
pp.insert(value: "Jim");
pp.insert(value: "Sergio");
pp.insert(value: "Alex");
pp.insert(value: "Jim");

assert_eq!(pp.contains(value: "Alex"), true);
assert_eq!(pp.contains(value: "Alexander"), false);
assert_eq!(pp.len(), 3);
```

Єдина задача, яку вирішує HashSet, це **унікальність** елементів



# Множина. HashSet<A>

Але порядок може бути абсолютно різний

```
let mut pp : HashSet<&str> = HashSet::new();
pp.insert(value: "Alex");
pp.insert(value: "Jim");
pp.iter() : impl Iterator<Item=&&str>
    .for_each(|x : &&str| println!("{x}")); // Jim, Alex

pp.insert(value: "Sergio");
pp.iter() : impl Iterator<Item=&&str>
    .for_each(|x : &&str| println!("{x}")); // Sergio, Jim, Alex
```

Це тому, що HashSet має оптимізовану структуру для знаходження дублікатів, і "порядок" технічно не має сенсу.

# Dictionary. HashMap<K, V> (Довідник ?)

HashMap — це таблиця "ключ - значення",  
Структура HashMap дуже схожа на структуру HashSet.  
Ключі мають бути унікальні.

Типи **K**, **V** - можуть бути абсолютно довільні  
Єдине обмеження,  
для типу **K** має бути імплементація **Hash**  
тобто можливість побудувати хеш-функцію  
Доречі, те саме стосується і **HashSet**



# HashMap. Приклади

На якій сторінці  
знаходиться розділ

Ключ	Значення
Розділ1	12
Розділ2	35
Розділ3	46
Розділ4	53

# HashMap. Приклади

Яка кількість слів у тексті

Ключ	Значення
"hello"	1
"to"	20
"be"	10
"learn"	15



# HashMap. Приклади Коду

```
let mut m: HashMap<&str, i32> = HashMap::new();  
m.insert(k: "hello", v: 1);  
m.insert(k: "to", v: 20);  
println!("{m:?}"); // {"to": 20, "hello": 1}  
  
assert_eq!(m.get(k: &"hello"), Some(&1));  
assert_eq!(m.contains_key(k: "hello"), true);  
assert_eq!(m.len(), 2);
```

Базові методи ті самі: **len**, **insert**, **get**, **contains**

# HashMap. Приклади коду

HashMap має дуже зручний API для модифікації.  
Наприклад ми рахуємо кількість слів.  
Нам потрібно перевірити, чи існує слово в HashMap.  
Якщо не існує — вставити з показником кількості 1.  
Якщо існує — то збільшити значення на 1.

```
let mut m: HashMap<&str, i32> = HashMap::new();  
  
m.entry(key: "to") : Entry<&str, i32>  
    .and_modify(|v| &mut i32 | *v += 1) : Entry<&str, i32>  
    .or_insert(default: 1);
```



# Задача підрахунку кількості літер

```
let xs : HashMap<char, i32> = "learning rust is an interesting process"
    .chars() : impl Iterator<Item=char>
    .fold(HashMap::new(), f: |mut m : HashMap<char, i32>, c : char| {
        m.entry(c) : Entry<char, i32>
            .and_modify(|v : &mut i32| *v += 1) : Entry<char, i32>
            .or_insert(default: 1);
        return m;
    });
println!("{xs:?}")
// {'c': 1, 'a': 2, 'p': 1, 'i': 4, 'n': 5, 'r': 4, 'e': 4,
//  'l': 1, ' ': 5, 's': 5, 't': 3, 'o': 1, 'u': 1, 'g': 2}
```

Наголошуємо, що ми не знаємо в якому порядку будуть надруковані дані, оскільки внутрішня структура оптимізована

**Але інколи,  
нас цікавить порядок,  
в якому зберігаються дані  
та деякі інші речі**



# Множина TreeSet<A>

Як ми знаємо, Set гарантує унікальність елементів

```
let mut xs : BTreeSet<i32> = BTreeSet::new();

xs.insert(value: 1);
xs.insert(value: 3);
xs.insert(value: 3);
xs.insert(value: 5);
xs.insert(value: 7);
xs.insert(value: 8);

assert_eq!(xs.len(), 5);
assert_eq!(xs.contains(value: &6), false);
```

Звісно ми знаємо що **6** не існує,

```
// ..  
xs.insert(value: 5);  
xs.insert(value: 7);  
// ..  
assert_eq!(xs.contains(value: &6), false);
```

Але ми “хочемо” знати які “найближчі” до цього елементи існують.  
BTreeSet вирішує цю проблему

```
let lower : Option<i32> = xs.range(..6).next_back();  
let upper : Option<i32> = xs.range(6..).next();  
assert_eq!(lower, Some(&5));  
assert_eq!(upper, Some(&7));
```



Також BTreeSet вирішує проблему мінімального елемента і максимального елемента в множині

```
let xs : BTreeSet<i32> =  
    BTreeSet::from(arr: [1, 2, 3, 5, 7, 9, 11]);  
  
assert_eq!(xs.first(), Some(&1));  
assert_eq!(xs.last(), Some(&11));
```

Як ми бачимо, функції мають назву **first/last** тобто ми маємо Ordering!

```
let xs : BTreeSet<i32> =  
    BTreeSet::from(arr: [11, 2, 7, 5, 3, 9, 1]);  
  
xs.iter() : impl Iterator<Item=&i32>  
    .for_each(|x : &i32| print!("{x}, "));  
// 1, 2, 3, 5, 7, 9, 11,
```

# BTreeMap<K, V>

BTreeMap поєднує функціонал BTreeSet і HashMap

Тобто коли ми “збираємо” дані в мапу, вони автоматично “сортуються”. Але не зовсім.

Ітератор знає як їх видавати у відсортованому вигляді



# BTreeMap. Приклади

```
let xs : BTreeMap<char, i32> = "learning rust is an interesting process"
    .chars() : impl Iterator<Item=char>
    .fold(BTreeMap::new(), f: |mut m : BTreeMap<char, i32>, c : char| {
        m.entry(c) : Entry<...>
            .and_modify(|v : &mut i32| *v += 1) : Entry<...>
            .or_insert(default: 1);
        return m;
    });
println!("{xs:?}")
// {' ': 5, 'a': 2, 'c': 1, 'e': 4, 'g': 2, 'i': 4, 'l': 1,
//  'n': 5, 'o': 1, 'p': 1, 'r': 4, 's': 5, 't': 3, 'u': 1}
```

# Висновок

Rust пропонує потужні інструменти для роботи з колекціями, кожна з яких має свої особливості.

Використання правильної колекції дозволяє ефективно організовувати та обробляти дані, забезпечуючи високу продуктивність і безпеку коду, фокусуючись на речах які потрібно імплементувати а не на деталях, де можливо допустити помилку.



**Код з лекцій,  
презентації Keypnote,  
PDF-файли  
знаходяться на GitHub:**

<https://github.com/djnzx/rust-course>  
<git@github.com:dnzxr/rust-course.git>