Лекція 23.
Послідовні контейнери.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Деки (Deque)
2. Список (List)
3. Однозв’язний список (Forward List)
4. Загальні характеристики послідовних контейнерів

Деки (Deque).

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Деки (Deque)

Дек — це двостороння черга (**double-ended queue**), яка дозволяє ефективно додавати і видаляти елементи як з початку, так і з кінця. Контейнер deque — це динамічна структура даних, подібна до vector, але з розширеною гнучкістю для операцій з обох боків.

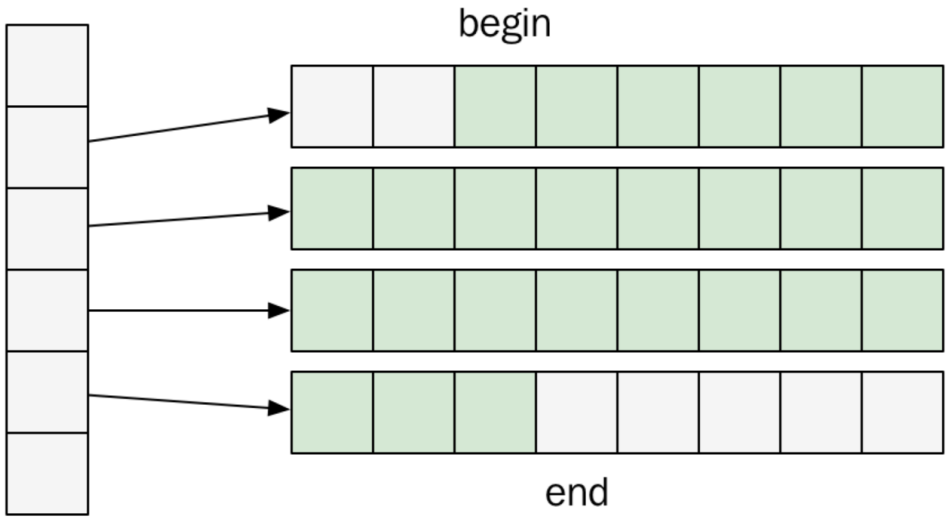
Основні особливості deque:

- Фіксований час доступу до елементів за індексом (O(1))
- Швидке додавання та видалення елементів з початку і з кінця
- Підтримка стандартних методів STL (size(), at(), begin(), end() тощо)
- Не гарантується неперервність зберігання елементів у пам’яті

Для того щоб використовувати деки, потрібно підключити #include <deque>

Внутрішня структура Deque

На відміну від vector, який зберігає дані в одному суцільному блоці пам’яті, deque зберігає дані в кількох невеликих блоках (chunks), і використовує таблицю вказівників на ці блоки.



Порівняння std::vector vs std::deque

Ознака	`std::vector`	`std::deque`
Тип пам’яті	Суцільна (один блок)	Сегментована (набори блоків)
Доступ за індексом `[]`	✅ O(1), дуже швидкий	✅ O(1), трохи повільніше через обчислення
`push_back()`	✅ O(1) амортизовано	✅ O(1) амортизовано
`push_front()`	❌ O(n) (через зсув усіх елементів)	✅ O(1) амортизовано
`insert()` **в середину**	⚠️ O(n)	⚠️ O(n)
`erase()` **в середині**	⚠️ O(n)	⚠️ O(n)

Основні методи

https://en.cppreference.com/w/cpp/container/deque

`void assign(size_type count, const T& value)`	Присвоює count копій значення value
`void swap(deque& other)`	Обмінює вміст із іншим контейнером
`T& at(size_type pos)`	Доступ до елемента з перевіркою меж
`T& operator[](size_type pos)`	Доступ до елемента без перевірки
`T& front()`	Повертає перший елемент
`T& back()`	Повертає останній елемент
`void push_back(const T& value)`	Додає елемент у кінець
`void push_front(const T& value)`	Додає елемент на початок
`void pop_back()`	Видаляє останній елемент
`void pop_front()`	Видаляє перший елемент

Основні методи

`iterator insert(iterator pos, const T& value)`	Вставляє елемент перед pos
`void clear()`	Очищає контейнер
`void resize(size_type count)`	Змінює розмір
`void shrink_to_fit()`	Звільняє зайву пам’ять
`bool empty() const`	Перевіряє, чи порожній
`size_type size() const`	Кількість елементів
`size_type max_size() const`	Максимальна кількість елементів
`iterator begin()`	Ітератор на початок
`iterator end()`	Ітератор на кінець

Основні методи

`reverse_iterator rbegin()`	Реверсний ітератор на кінець
`reverse_iterator rend()`	Реверсний ітератор на початок
`const_iterator cbegin() const`	Константний ітератор на початок
`const_iterator cend() const`	Константний ітератор на кінець
`const_reverse_iterator crbegin() const`	Константний реверсний ітератор на кінець
`const_reverse_iterator crend() const`	Константний реверсний ітератор на початок

Швидкодія

Опис	Складність
Доступ до елемента за індексом (at(), operator[])	O(1)
Додавання елемента в кінець (push_back())	O(1)
Додавання елемента на початок (push_front())	O(1)
Видалення останнього елемента (pop_back())	O(1)
Видалення першого елемента (pop_front())	O(1)
Вставка елемента всередину (insert())	O(n)
Видалення елемента всередині (erase())	O(n)
Очищення всіх елементів (clear())	O(n)
Перевірка, чи контейнер порожній (empty())	O(1)

Додавання з обох боків

Методи push_front() і push_back() дозволяють додавати елементи на початок і в кінець deque. Ці операції виконуються ефективно — без зміщення інших елементів.

```
int main() {
    deque<int> numbers;
    numbers.push_back(10);
    numbers.push_front(5);
    numbers.push_back(15);
    numbers.push_front(1);
    for (int number : numbers) cout << number << " ";
    // Результат: 1 5 10 15
}
```

Видалення з обох боків

Методи pop_front() та pop_back() дозволяють швидко видаляти елементи з початку та кінця. Ідеально для реалізації черг або двосторонніх буферів.

```
int main() {
    deque<int> numbers{1, 2, 3, 4, 5};
    numbers.pop_front();  // видаляє 1
    numbers.pop_back();   // видаляє 5
    for (int number : numbers) cout << number << " ";
    // Результат: 2 3 4
}
```

Доступ до елементів

deque підтримує доступ за індексом через [] і at(), а також доступ до першого й останнього елемента.
Метод at() перевіряє межі, [] — ні.

```
int main() {
    deque<string> letters{"A", "B", "C"};
    cout << letters[0] << "\n";     // A
    cout << letters.at(1) << "\n";  // B
    cout << letters.back() << "\n"; // C
    cout << letters.front() << "\n";// A
}
```

Вставка в середину

Метод insert() вставляє елемент у вказану позицію.
На відміну від push_back() і push_front(), має складність O(n) через можливе зміщення елементів.

```
int main() {
    deque<int> numbers{10, 20, 30};
    numbers.insert(numbers.begin() + 1, 15); // вставляє 15 перед 20
    for (int number : numbers) cout << number << " ";
    // Результат: 10 15 20 30
}
```

Очищення контейнера

Метод clear() повністю видаляє всі елементи з deque, але не змінює capacity.
Після виклику empty() поверне true.

```
int main() {
    deque<int> numbers{1, 2, 3};
    numbers.clear();
    cout << "Size: " << numbers.size() << "\n";
    cout << "Is empty: " << boolalpha << numbers.empty() << "\n";
    // Результат:
    // Size: 0
    // Is empty: true
}
```

Зміна розміру

Метод resize(n) змінює кількість елементів у deque.
Якщо n > size(), нові елементи ініціалізуються значенням за замовчуванням (0 для int).

```
int main() {
    deque<int> numbers{1, 2, 3};
    numbers.resize(6);
    for (int number : numbers) cout << number << " ";
    // Результат: 1 2 3 0 0 0
}
```

Список (List).

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Список (List)

Список — це двозв'язний список (**doubly-linked list**), який дозволяє ефективно вставляти і видаляти елементи в будь-якому місці контейнера.
Контейнер list — це послідовна структура даних, не підтримує доступ за індексом, але забезпечує стабільність ітераторів та швидке переміщення елементів.

Основні особливості list:

- Швидке додавання та видалення елементів у будь-якій позиції (O(1))
- Відсутність доступу за індексом ([] не підтримується)
- Підтримка стандартних методів STL (push_back(), insert(), erase() тощо)
- Гарантована стабільність ітераторів після вставки або видалення

Для того щоб використовувати списки , потрібно підключити #include <list>

Основні методи

https://en.cppreference.com/w/cpp/container/list

`T& front()`	Повертає перший елемент
`T& back()`	Повертає останній елемент
`void push_front(const T& value)`	Додає елемент на початок
`void push_back(const T& value)`	Додає елемент у кінець
`void pop_front()`	Видаляє перший елемент
`void pop_back()`	Видаляє останній елемент

Основні методи

`iterator insert(iterator pos, const T& value)`	Вставляє елемент перед позицією pos
`iterator erase(iterator pos)`	Видаляє елемент за ітератором
`void assign(size_type count, const T& value)`	Присвоює count копій значення
`void clear()`	Видаляє всі елементи зі списку
`void resize(size_type count)`	Змінює розмір списку

Основні методи

`void swap(list& other)`	Обмінює вміст із іншим списком
`void merge(list& other)`	Об’єднання впорядкованих списків на основі операції “<”
`void splice(iterator pos, list& other)`	Переміщує всі елементи other у позицію pos
`void remove(const T& value)`	Видаляє всі елементи, рівні value
`void unique()`	Видаляє послідовні дублікати
`void reverse()`	Змінює порядок елементів на зворотній
`void sort()`	Сортує елементи списку

Основні методи

`bool empty() const`	Перевіряє, чи список порожній
`size_type size() const`	Повертає кількість елементів
`size_type max_size() const`	Повертає максимально можливу кількість елементів
`iterator begin()`	Ітератор на перший елемент
`iterator end()`	Ітератор на елемент після останнього
`const_iterator cbegin() const`	Константний ітератор на початок
`const_iterator cend() const`	Константний ітератор на кінець
`reverse_iterator rbegin()`	Реверсний ітератор на останній елемент
`reverse_iterator rend()`	Реверсний ітератор перед першим елементом
`const_reverse_iterator crbegin() const`	Константний реверсний ітератор на кінець
`const_reverse_iterator crend() const`	Константний реверсний ітератор на початок

Швидкодія

Опис	Складність
Додавання / видалення з початку або кінця (push_front(), push_back(), pop_front(), pop_back())	O(1)
Вставка або видалення за ітератором (insert(), erase())	O(1)
Перевірка порожнечі (empty()) / доступ до країв (front(), back())	O(1)
Пошук елемента / видалення за значенням (remove())	O(n)
Реверс (reverse()), видалення дублікатів (unique())	O(n)
Сортування (sort())	O(n log n)
Очищення (clear())	O(n)
Доступ за індексом (немає, тільки через ітератор або std::advance)	O(n)

Додавання і видалення з обох боків

Методи push_front(), push_back(), pop_front() та pop_back() дозволяють ефективно додавати і видаляти елементи з обох боків списку.
У list це виконується за O(1) без зміщення елементів.

```
int main() {
    list<int> numbers;
    numbers.push_back(10);
    numbers.push_front(5);
    numbers.push_back(15);
    numbers.push_front(1);
    numbers.pop_back();
    for (int number : numbers) cout << number << " ";
    // Результат: 1 5 10
}
```

Вставка і видалення за ітератором

list дозволяє ефективно вставляти та видаляти елементи у будь-якій позиції, використовуючи ітератори.

```
int main() {
    list<int> numbers{10, 30};
    auto it = next(numbers.begin());
    numbers.insert(it, 20);
    numbers.erase(numbers.begin());
    for (int number : numbers) cout << number << " ";
    // Результат: 20 30
}
```

Очищення та перевірка

Метод clear() очищає список, а empty() дозволяє перевірити, чи він порожній.

```
int main() {
    list<int> numbers{1, 2, 3};
    numbers.clear();
    cout << "Size: " << numbers.size() << " ";
    cout << "Empty: " << boolalpha << numbers.empty();
    // Результат: Size: 0 Empty: true
}
```

Сортування і унікальність

list підтримує методи sort() і unique(), що дозволяють відсортувати список і видалити дублікати.

```
int main() {
    list<int> numbers{3, 1, 2, 2, 3};
    numbers.sort();
    numbers.unique();
    for (int number : numbers) cout << number << " ";
    // Результат: 1 2 3
}
```

Об’єднання списків

Метод splice() переносить елементи з одного списку в інший без копіювання.
Це можливо лише в list завдяки зв’язаній структурі.

```
int main() {
    list<int> a{1, 2};
    list<int> b{3, 4};
    a.splice(a.end(), b); // вставляє всі елементи b в кінець a
    for (int x : a) cout << x << " ";
    // Результат: 1 2 3 4
}
```

Об’єднання відсортованих списків

merge() об’єднує два відсортовані списки в один.
Ефективно і без додаткових алокацій.

```
int main() {
    list<int> a{1, 3, 5};
    list<int> b{2, 4, 6};
    a.merge(b);
    for (int x : a) cout << x << " ";
    // Результат: 1 2 3 4 5 6
}
```

Однозв’язний список (Forward List).

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Однозв’язний список (Forward List)

Однозв’язний список — це структура даних (**singly-linked list**), у якій кожен елемент містить посилання лише на наступний.
Контейнер forward_list — це легкий варіант list, який забезпечує ефективне вставлення та видалення, але не підтримує доступу до попереднього елемента або випадкового доступу.

Основні особливості forward_list:

- Ефективне додавання і видалення лише після вказаної позиції
- Мінімальні витрати пам’яті порівняно з list (одне посилання замість двох)
- Відсутність доступу за індексом (немає [] і at())
- Не підтримує size() (на відміну від інших контейнерів)
- Лише односторонні ітератори (не підтримує rbegin(), end() - 1 тощо)

Для того щоб використовувати forward list, потрібно підключити #include <forward_list>

Основні методи

https://en.cppreference.com/w/cpp/container/forward_list

`T& front()`	Повертає перший елемент
`void push_front(const T& value)`	Додає елемент на початок
`void pop_front()`	Видаляє перший елемент
`void assign(size_type count, const T& value)`	Присвоює count копій значення
`void clear()`	Видаляє всі елементи
`void resize(size_type count)`	Змінює розмір списку
`bool empty() const`	Перевіряє, чи список порожній

Основні методи

`iterator insert_after(iterator pos, const T& value)`	Вставляє елемент після pos
`iterator insert_after(iterator pos, size_type count, const T& value)`	Вставляє count копій значення
`iterator erase_after(iterator pos)`	Видаляє елемент після позиції
`iterator erase_after(iterator first, iterator last)`	Видаляє діапазон елементів після first до last

Основні методи

`void swap(forward_list& other)`	Обмінює вміст з іншим forward_list
`void merge(forward_list& other)`	Зливає відсортований список other
`void splice_after(iterator pos, forward_list& other)`	Переміщує всі елементи other після pos
`void remove(const T& value)`	Видаляє всі елементи, рівні value
`void unique()`	Видаляє послідовні дублікати
`void reverse()`	Змінює порядок елементів
`void sort()`	Сортує елементи списку

Основні методи

`iterator before_begin()`	Ітератор перед першим елементом
`const_iterator before_begin() const`	Константний ітератор перед першим елементом
`iterator begin()`	Ітератор на перший елемент
`iterator end()`	Ітератор на кінець
`const_iterator cbegin() const`	Константний ітератор на початок
`const_iterator cend() const`	Константний ітератор на кінець

Швидкодія

Опис	Складність
Додавання / видалення з початку (push_front(), pop_front())	O(1)
Вставка / видалення після позиції (insert_after(), erase_after())	O(1)
Перевірка порожнечі (empty()), доступ до початку (front())	O(1)
Пошук і видалення за значенням (remove())	O(n)
Видалення дублікатів (unique()), реверс (reverse())	O(n)
Сортування (sort()), злиття (merge())	O(n log n)
Очищення (clear())	O(n)
Зміна розміру (resize()), присвоєння (assign())	O(n)
Доступ за індексом (не підтримується, лише через advance())	O(n)

Додавання і видалення з початку

Методи push_front() та pop_front() дозволяють ефективно додавати і видаляти елементи лише з початку списку.
У forward_list ці операції виконуються за O(1), а доступу до кінця не передбачено.

```
int main() {
    forward_list<int> numbers;
    numbers.push_front(10);
    numbers.push_front(5);
    numbers.push_front(1);
    numbers.pop_front();
    for (int number : numbers) cout << number << " ";
    // Результат: 5 10
}
```

Вставка після елемента

Метод insert_after() дозволяє вставити елемент після вказаної позиції.
Це характерно лише для forward_list.

```
int main() {
    forward_list<int> numbers{1, 3};
    auto it = numbers.begin(); // вказує на 1
    numbers.insert_after(it, 2); // вставляємо 2 після 1
    for (int number : numbers) cout << number << " ";
    // Результат: 1 2 3
}
```

Видалення після елемента

Метод erase_after() видаляє елемент, що йде після вказаного.
Це дозволяє видаляти елементи без зсуву решти.

```
int main() {
    forward_list<int> numbers{1, 2, 3};
    auto it = numbers.begin(); // вказує на 1
    numbers.erase_after(it);   // видаляємо 2
    for (int number : numbers) cout << number << " ";
    // Результат: 1 3
}
```

Об’єднання відсортованих списків (merge())

Метод merge() дозволяє об'єднати відсортовані списки без копіювання.
Це характерна особливість зв’язаних списків.

```
int main() {
    forward_list<int> a{1, 3};
    forward_list<int> b{2, 4};
    a.merge(b);
    for (int number : a) cout << number << " ";
    // Результат: 1 2 3 4
}
```

Загальні характеристики послідовних контейнерів

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Загальні характеристики

Послідовні контейнери зберігають елементи в заданому порядку.
Вибір контейнера залежить від типу доступу, частоти вставок/видалень і обмежень по пам’яті.

Контейнер	Доступ за індексом	Додавання на початок	Додавання в кінець	Вставка в середину	Розмір
array	✅ O(1)	❌	❌	❌	фікс.
vector	✅ O(1)	❌	✅ O(1) амортизоване	❌ / повільно	змін.
deque	✅ O(1)	✅ O(1)	✅ O(1)	❌ / повільно	змін.
list	❌ O(n)	✅ O(1)	✅ O(1)	✅ O(1)	змін.
forward_list	❌ O(n)	✅ O(1)	❌	✅ O(1) після pos	змін.

Що коли обирати?

- array — коли відомий фіксований розмір на момент компіляції; мінімальні витрати, швидкий доступ.
- vector — динамічний масив з швидким доступом за індексом; найкращий для додавання в кінець у випадку наперед зарезервованого розміру.
- deque — швидке додавання і видалення з обох боків; доступ за індексом повільніший, ніж у vector.
- list — часті вставки/видалення всередині; немає доступу за індексом; вища вартість по пам’яті.
- forward_list — однозв’язаний список; мінімальні витрати пам’яті; тільки вставка/видалення після позиції.

Що коли обирати?

Задача: потрібно зберігати дані фіксованого розміру, який відомий наперед і не змінюється.
Контейнер: array — найменші витрати, найшвидший доступ, зберігається в стеку.

Задача: потрібно поступово додавати елементи в кінець і швидко звертатись до будь-якого за індексом.
Контейнер: vector — оптимальний для динамічного масиву, швидкий доступ і ефективне зростання.

Задача: потрібно додавати або видаляти елементи як з початку, так і з кінця.
Контейнер: deque — забезпечує швидке вставлення і видалення з обох боків.

Що коли обирати?

Задача: часто потрібно вставляти або видаляти елементи в середині колекції.
Контейнер: list — подвійно зв’язаний список, ефективна робота з ітераторами, стабільність при зміні.

Задача: потрібна максимально легка структура для вставок після елемента, без доступу назад.
Контейнер: forward_list — однозв’язаний список, мінімальні витрати пам’яті.

Приклад: Модель текстового редактора

Потрібно розробити спрощену модель текстового редактора, де користувач може редагувати текст по рядках:

- ❏ Швидко вставляти та видаляти рядки в будь-якому місці (наприклад, вставити новий рядок перед або після поточного).
- ❏ Переходити курсором вгору/вниз по тексту (по рядках).
- ❏ Зберігати порядок доданих рядків.

Приклад: Модель текстового редактора

- List - найоптимальніший контейнер для зберігання рядків тексту
- Cursor - ітератор, який вказує на поточний активний рядок

```cpp
// ==== Class declaration ====
class TextEditor {
private:
    std::list<std::string> lines;
    std::list<std::string>::iterator cursor;

public:
    TextEditor();

    void insertLineBelow(const std::string& line);
    void insertLineAbove(const std::string& line);
    void deleteCurrentLine();
    void moveCursorUp();
    void moveCursorDown();
    void printDocument() const;
    void printCurrentLine() const;
};
```

```cpp
#include "TextEditor.h"

// ==== Example usage ====
int main() {
    TextEditor editor;

    editor.insertLineBelow("First line");
    editor.insertLineBelow("Second line");
    editor.insertLineBelow("Third line");

    editor.printDocument();

    editor.moveCursorUp();  // Move to second line
    editor.insertLineAbove("Inserted before second");

    editor.printDocument();

    editor.deleteCurrentLine();
    editor.printDocument();

    return 0;
}
```

Приклад: Обчислення середнього

Задача:
Розробити клас **MovingAverage**, який у режимі реального часу дозволяє обчислювати середнє значення останніх *N* чисел, що надходять у вигляді потоку даних (з файлу або клавіатури). Клас повинен зберігати лише останні *N* значень, автоматично видаляючи найстаріше значення при додаванні нового, якщо розмір вікна перевищено.

Мета:
Забезпечити ефективне додавання нових значень та миттєве обчислення середнього значення поточного вікна.

Приклад: Обчислення середнього

```
class MovingAverage {
private:
    std::deque<int> window;
    size_t maxSize;
    double sum;

public:
    MovingAverage(size_t size);

    void add(int value);
    double average() const;
    void printWindow() const;
};
```

```
int main() {
    MovingAverage avrgWindow(3);

    avrgWindow.add(10);
    avrgWindow.printWindow();
    std::cout << "Average: " << avrgWindow.average() << "\n\n";

    avrgWindow.add(20);
    avrgWindow.printWindow();
    std::cout << "Average: " << avrgWindow.average() << "\n\n";

    avrgWindow.add(30);
    avrgWindow.printWindow();
    std::cout << "Average: " << avrgWindow.average() << "\n\n";

    avrgWindow.add(40);
    avrgWindow.printWindow();
    std::cout << "Average: " << avrgWindow.average() << "\n";

    return 0;
}
```

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY