

Лекція 22.

Стандартна бібліотека шаблонів (STL).

Послідовні контейнери



План на сьогодні

1

Компоненти бібліотеки STL

2

Загальна модель контейнерів та основні завдання

3

Види контейнерів

4

Послідовні контейнери

5

Інтерфейсні та ітераторні типи послідовних контейнерів

6

Масив та вектор



Компоненти бібліотеки STL



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Рекомендовані онлайн курси

- ❑ <https://ua.udemy.com/course/beginning-c-plus-plus-programming/>
- ❑ <https://ua.udemy.com/course/cpp-deep-dive/>
- ❑ <https://ua.udemy.com/course/the-complete-cpp-developer-course/>
- ❑ <https://ua.udemy.com/course/competitive-programming-algorithms-coding-minutes/>
- ❑ <https://ua.udemy.com/course/skills-algorithms-cpp/>

Література

- ❑ [C++17 STL Cookbook \(O'Reilly\)](#)
- ❑ [The C++ Standard Library \(Leanpub\)](#)
- ❑ [Effective Modern C++ \(O'Reilly\)](#)
- ❑ [Modern C++ Programming Cookbook - Second Edition \(O'Reilly\)](#)
- ❑ [C++ High Performance - Second Edition \(O'Reilly\)](#)

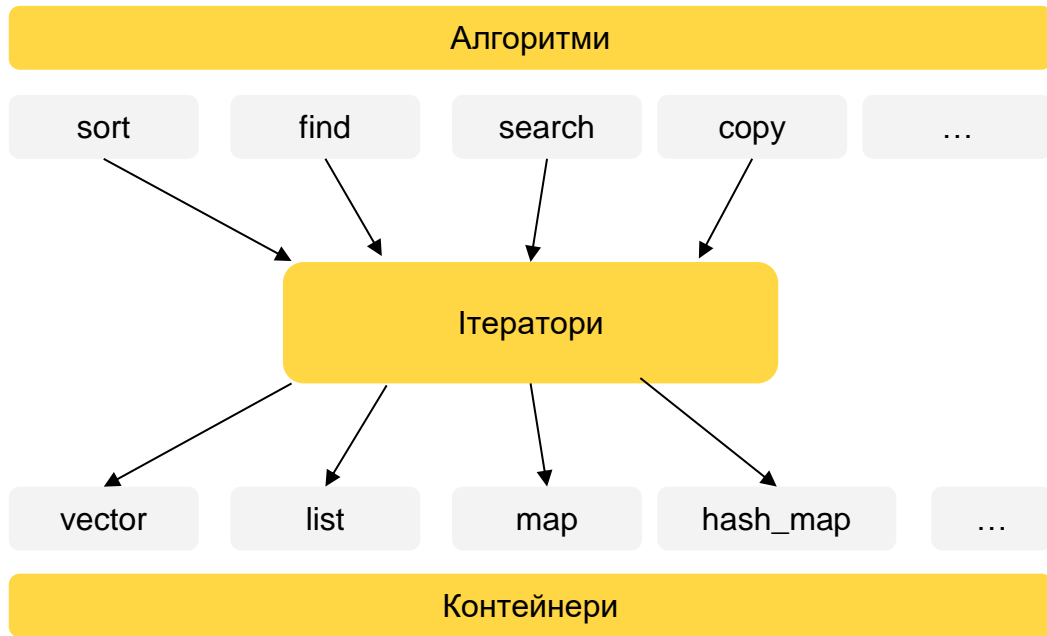
Компоненти бібліотеки STL

- ❑ **Стандартна бібліотека шаблонів C++ (STL)** — це набір шаблонних класів і функцій, що забезпечує реалізацію поширених структур даних і алгоритмів, таких як списки, стеки, масиви, сортування, пошук тощо. Вона також надає ітератори та функтори, які спрощують роботу з алгоритмами та контейнерами.

Основні компоненти

- ❑ **Контейнери** - структури для зберігання даних
- ❑ **Алгоритми** - глобальні функції для обробки даних;
- ❑ **Ітератори** - засоби, які володіють методами для обходу контейнерів і доступу до даних;
- ❑ **Функтори (об'єкти функції)** - класи, в яких перевантажено оператор виклику функції; використання об'єктів подібне до виклику функції;
- ❑ **Адаптери** - модифікатори інтерфейсу компонент

Основна модель



Розділення учасників

Алгоритми маніпулюють даними, але не знають про контейнери

Контейнери зберігають дані, але не знають про алгоритми

Алгоритми і контейнери взаємодіють через **ітератори**

Кожен контейнер має свої власні ітераторні типи

Основні завдання

- ☐ Зберігання даних в контейнері
- ☐ Організація даних
 - ☐ Для роздруку
 - ☐ Для швидкого доступу
- ☐ Отримання даних
 - ☐ За індексом (e.g., отримати N-ий елемент)
 - ☐ За значенням (e.g., отримати перший елемент зі значенням "Chocolate")
 - ☐ За властивістю (e.g., отримати перший елемент для якого "age<64")
- ☐ Додавання та видалення даних
- ☐ Сортуння і пошук
- ☐ Прості числові операції

Види контейнерів

Послідовні контейнери

Контейнери-послідовності (sequence containers) передбачають, що у кожний поточний момент існує певний порядок слідування елементів у контейнері

- ❑ елементами є безпосередньо об'єкти
- ❑ послідовний обхід всіх елементів контейнера
- ❑ передача підпослідовностей в алгоритми для обробки
- ❑ рекомендованим засобом для перебору елементів є ітератор, який і забезпечує потрібний напрямок переміщень в контейнері

Послідовні контейнери

- ❑ Послідовні контейнери зберігають дані лінійно. Вони також використовуються для реалізації адаптерів контейнерів

У C++ STL існує 5 послідовних контейнерів

Масиви (**Arrays**)

Вектори (**Vector**)

Двостороння
черга (**Deque**)

Списки
(**List**)

Однозв'язні
списки
(**Forward List**)

Впорядковані асоціативні контейнери

Ordered Associative Containers

- ☐ елементами є пари ключ-об'єкт
- ☐ структура зберігання – збалансоване бінарне пошукове дерево
- ☐ завжди відсортований контейнер за значенням ключа
- ☐ доступ за значенням ключа
- ☐ Складність пошуку, вставки та видалення $O(\log n)$

Відображення
(асоціативний масив)
map

Мультівідображення
multimap

Множина
set

Мультимножина
multiset

Ключ:

- ☐ набуває значення в межах будь-якого типу, який допускає впорядкування (визначений оператор порівняння “<”)
- ☐ у відображеннях значення ключа не може повторюватися
- ☐ У мультівідображеннях можуть зберігатися пари з однаковими значеннями ключа

Невпорядковані асоціативні контейнери

Unordered Associative Containers

- ❑ елементами є пари ключ-об'єкт
- ❑ структура зберігання – геш-таблиця
- ❑ невпорядкований контейнер
- ❑ доступ за значенням ключа ($\text{bucket_index} = \text{hash_function}(\text{key}) \bmod \text{num_buckets}$)
- ❑ Складність пошуку, вставки та видалення $O(1)$

Відображення
`unordered_map`

Мультивідображення
`unordered_multimap`

Множина
`unordered_set`

Мультимножина
`unordered_multiset`

Ключ:

- ❑ набуває значення в межах будь-якого типу, для якого визначені оператор порівняння (`==`) та функція гешування
- ❑ у відображеннях значення ключа не може повторюватися
- ❑ У мультивідображеннях можуть зберігатися пари з однаковими значеннями ключа

Контейнери адаптери

Контейнери-адаптери — це обгортки (wrapper) над стандартними контейнерами STL (`vector`, `deque`, `list`), які обмежують доступ до даних і надають спеціалізовані методи роботи. Вони реалізують черги, стеки та черги з пріоритетом.

Основна ідея: адаптери не є контейнерами самі по собі, вони використовують інший контейнер у якості бази та надають специфічний інтерфейс.

Стек
`Stack`
(`deque` by default)

Черга
`Queue`
(`deque` by default)

Черга з пріоритетом
`priority_queue`
(`vector` by default)

Інтерфейсні типи послідовних контейнерів



Інтерфейсні типи послідовних контейнерів

Ідентифікація тип	Тип компонента
<code>value_type</code>	Тип елемента як одиниці зберігання в контейнері
<code>allocator_type</code>	Тип менеджера пам'яті
<code>size_type</code>	Тип для представлення розміру елемента і кількості елементів у контейнері
<code>difference_type</code>	Тип для представлення різниці адрес елементів контейнера, які повернули два ітератори
<code>reference</code>	Посилання на елемент контейнера
<code>const_reference</code>	Посилання на елемент контейнера, яке не допускає його модифікації
<code>pointer</code>	Вказівник на елемент контейнера
<code>const_pointer</code>	Вказівник на елемент контейнера, який не допускає його модифікації

Ітераторні типи послідовних контейнерів

Ідентифікатор	Тип компонента
<code>iterator</code>	Ітератор для перебору елементів контейнера
<code>const_iterator</code>	Ітератор для перебору елементів контейнера, який не допускає їхньої зміни
<code>reverse_iterator</code>	Ітератор для перебору елементів контейнера згідно порядку, зворотнього до звичайного
<code>const_reverse_iterator</code>	Ітератор для перебору без модифікації елементів контейнера згідно порядку, зворотнього до звичайного

Методи доступу до ітераторів

Метод	Тип ітератора
<code>iterator begin();</code> <code>const_iterator begin() const;</code>	Ітератор, настроєний на перший елемент контейнера.
<code>iterator end();</code> <code>const_iterator end() const;</code>	Ітератор зі значенням past-the-end.
<code>reverse_iterator rbegin();</code> <code>const_reverse_iterator rbegin() const;</code>	Зворотній ітератор зі значенням past-the-end.
<code>reverse_iterator rend();</code> <code>const_reverse_iterator rend() const;</code>	Зворотній ітератор, настроєний на перший елемент контейнера.

Методи доступу до елементів

Метод	Дія методу
<code>reference operator[] (size_type n; const_reference operator[] (size_type n) const;</code>	Доступ за індексом до елемента без перевірки виходу за межі діапазону контейнера (лише для <code>array</code> , <code>vector</code> та <code>deque</code>)
<code>reference at(size_type n); const_reference at(size_type n) const;</code>	Доступ за індексом до елемента з перевіркою виходу за межі діапазону контейнера (лише для <code>array</code> , <code>vector</code> та <code>deque</code>)
<code>reference front(); const_reference front() const;</code>	Доступ до першого елемента контейнера (<code>==begin()</code>)
<code>reference back(); const_reference back() const;</code>	Доступ до останнього елемента контейнера (<code>!=end()</code>)

Методи характеристик контейнерів

Метод	Дія методу
<code>bool empty() const;</code>	Повертає true , якщо розмір контейнера нульовий.
<code>size_type size() const;</code>	Повертає кількість елементів контейнера
<code>allocator_type get_allocator() const;</code>	Повертає копію розподільувача пам'яті.

Методи порівняння контейнерів

Метод	Результат операції
<pre>template <class T, class Alloc> inline bool operator==(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо контейнер x є таким, як y .
<pre>template <class T, class Alloc> inline bool operator!=(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо контейнер x відрізняється від y . (до C++ 20)
<pre>template <class T, class Alloc> inline bool operator<(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо x лексикографічно менший, ніж y . (до C++ 20)
<pre>template <class T, class Alloc> inline bool operator>(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо x лексикографічно більший, ніж y . (до C++ 20)
<pre>template <class T, class Alloc> inline bool operator<=(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо x лексикографічно менший/рівний, ніж y . (до C++ 20)
<pre>template <class T, class Alloc> inline bool operator>=(const cont<T,Alloc>& x, const cont<T,Alloc>& y);</pre>	true , якщо x лексикографічно більший/рівний, ніж y . (до C++ 20)

Масиви (Arrays)



Масиви (Arrays)

Масив — це колекція однорідних об'єктів, і контейнер `array` призначений для масивів із фіксованим розміром. Цей контейнер огортає статичний масив з фіксованим розміром (C-style array) та надає додатковий функціонал та безпеку.

Основні особливості `std::array`:

- ❑ Фіксований розмір (не змінюється після створення).
- ❑ Швидкий доступ до елементів ($O(1)$).
- ❑ Підтримка стандартних методів STL (`size()`, `at()`, `begin()`, `end()` тощо).
- ❑ Не вимагає динамічного розподілу пам'яті (зберігається в стеку).

Для того щоб використовувати масиви, потрібно підключити `#include <array>`:

```
std::array<тип, розмір> ім'я;
```

Основні методи

<https://en.cppreference.com/w/cpp/container/array>

Метод	Опис
<code>size()</code>	Повертає кількість елементів
<code>at(index)</code>	Доступ до елемента з перевіркою меж
<code>operator[]</code>	Доступ до елемента без перевірки меж
<code>front()</code>	Повертає перший елемент
<code>back()</code>	Повертає останній елемент
<code>data()</code>	Повертає вказівник на перший елемент
<code>begin()</code> , <code>end()</code>	Ітератори на початок і кінець
<code>rbegin()</code> , <code>rend()</code>	Реверсні ітератори
<code>fill(value)</code>	Заповнює всі елементи одним значенням
<code>swap(other_array)</code>	Міняє місцями два масиви однакового типу і розміру

Синтаксис std::array

Це структура, яка містить **вбудований C-style масив** (T elems[N];):

```
template <typename T, std::size_t N>
```

```
struct array {
```

```
    T elems[N]; // Масив фіксованого розміру
```

```
};
```

Приклад

```
// C++ program to demonstrate working of array
```

```
#include <algorithm>
```

```
#include <array>
```

```
#include <iostream>
```

```
#include <iterator>
```

```
using namespace std;
```

```
int main() {
```

```
    array<int, 5> ar1{3, 4, 5, 1, 2};
```

```
    array<int, 5> ar2 = {1, 2, 3, 4, 5};
```

```
    array<string, 2> ar3 = {"a", "b"};
```

```
    cout << "Sizes of arrays are" << endl;
```

```
    cout << ar1.size() << endl;
```

```
    cout << ar2.size() << endl;
```

```
    cout << ar3.size() << endl;
```

```
    cout << "\nInitial ar1 : ";
```

```
    for (auto i : ar1)
```

```
        cout << i << ' ';
```

```
    // container operations are supported
```

```
    sort(ar1.begin(), ar1.end());
```

```
    cout << "\nsorted ar1 : ";
```

```
    for (auto i : ar1)
```

```
        cout << i << ' ';
```

```
    // Filling ar2 with 10
```

```
    ar2.fill(10);
```

```
    cout << "\nFilled ar2 : ";
```

```
    for (auto i : ar2)
```

```
        cout << i << ' ';
```

```
    // ranged for loop is supported
```

```
    cout << "\nar3 : ";
```

```
    for (auto &s : ar3)
```

```
        cout << s << ' ';
```

```
    return 0;
```

```
}
```

Sizes of arrays are

5

5

2

Initial ar1 : 3 4 5 1 2

sorted ar1 : 1 2 3 4 5

Filled ar2 : 10 10 10 10 10

ar3 : a b

Оператор []

❏ **Оператор []:** Цей оператор працює так само, як і в звичайному масиві — використовується для доступу до елемента, що зберігається за індексом [i](#).

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<char, 3> arr={'G', 'f', 'G'};
    cout<< arr[0] <<" "<<arr[2];
    return 0;
}
```

A black rectangular box representing a terminal window. Inside, the text "G G" is displayed in a white, monospaced font, with a space between the two characters. This represents the output of the C++ program shown in the adjacent code block.

G G

Функції front() та back()

❏ Функції **front()** та **back()**: Ці методи використовуються для безпосереднього доступу до першого та останнього елемента масиву.

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<int , 3> arr={'G','f','G'}; // ASCII val of 'G' =71
    Cout << arr.front() << " " << arr.back();
    return 0;
}
```

71 71

Функція empty()

❏ Функція `empty()`: Ця функція використовується для перевірки, чи є оголошений масив STL порожнім. Якщо масив порожній — повертає `true`, інакше — `false`.

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array <int , 3> arr={'G','f','G'}; // ASCII val of
    'G' =71
    array <int , 3> arr1={'M','M','P'}; // ASCII val of
    'M' = 77 and 'P' = 80
    bool x = arr.empty(); // false ( not empty)
    cout<<boolalpha<<(x);
    return 0;
}
```

false

Функція at()

- ❑ **Функція `at()`**: Ця функція використовується для доступу до елемента, що зберігається на певній позиції в масиві. На відміну від оператора `[]`, якщо спробувати звернутися до елемента за межами допустимого діапазону індексів, `at()` згенерує виняток (`std::out_of_range`).

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<int, 3> arr={'G','f','G'}; // ASCII val of
    'G' =71
    array<int, 3> arr1={'M','M','P'}; // ASCII val of
    'M' = 77 and 'P' = 80
    cout<< arr.at(2) <<" " << arr1.at(2);
    //cout<< arr.at(3); // exception{Abort signal from
    abort(3) (SIGABRT)}
    return 0;
}
```



71 80

Функція fill()

- ❑ **Функція `fill()`:** Ця функція спеціально призначена для ініціалізації або заповнення всіх елементів масиву однаковим значенням.

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<int, 5> arr;
    arr.fill(1);
    for(int i: arr)
        cout<<arr[i]<<" ";
    return 0;
}
```



1 1 1 1 1

size(), max_size() та sizeof()

Функції `size()`, `max_size()` та `sizeof()`:

- ❑ `size()` / `max_size()` — обидві функції повертають кількість елементів, які може містити масив. У випадку `std::array` вони завжди повертають однакове значення, оскільки розмір масиву фіксований.
- ❑ `sizeof()` — використовується для отримання загального розміру масиву в байтах.

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<int, 10> arr;
    cout<<arr.size()<<'\n'; // total num of indexes
    cout<<arr.max_size()<<'\n'; // total num of indexes
    cout<<sizeof(arr); // total size of array
    return 0;
}
```

10

10

40

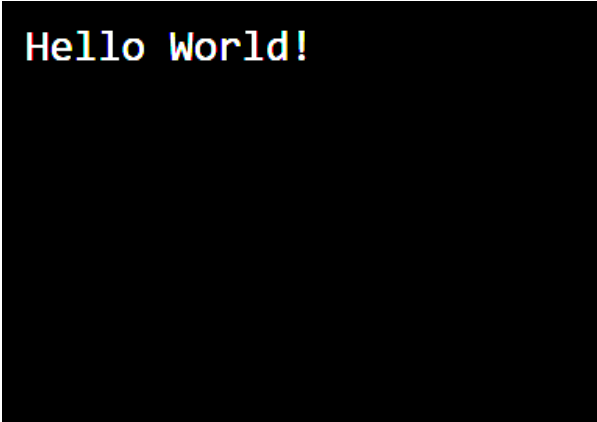
data()

- **data()**: Ця функція повертає вказівник на перший елемент об'єкта масиву. Оскільки елементи в масиві зберігаються у суміжних комірках пам'яті, функція **data()** повертає базову адресу об'єкта типу рядка/char.

```
#include <iostream>
#include <cstring>
#include <array>

using namespace std;

int main ()
{
    const char* str = "Hello World!";
    array<char,13> arr;
    memcpy (arr.data(),str,13);
    cout << arr.data() << '\n';
    return 0;
}
```



Hello World!

Вектор (Vector)

Вектори (Vectors)

❏ У C++ `vector` — це динамічний масив, який зберігає колекцію елементів одного типу в суміжній пам'яті. Він має здатність автоматично змінювати свій розмір під час додавання або видалення елементів.

❏ `Vector` визначається як шаблон класу `std::vector` у заголовковому файлі `<vector>`.

```
#include <vector>
```

<https://en.cppreference.com/w/cpp/container/vector>

Методи

push_back()	Додає елемент у кінець вектора.
pop_back()	Видаляє останній елемент вектора.
size()	Повертає кількість елементів у векторі.
max_size()	Повертає максимальну кількість елементів, яку може містити вектор.
resize()	Змінює розмір вектора.
empty()	Перевіряє, чи є вектор порожнім.
operator[]	Доступ до елемента за індексом (без перевірки меж).
at()	Доступ до елемента за індексом з перевіркою меж.
front()	Повертає перший елемент вектора.
back()	Повертає останній елемент вектора.

Методи

begin()	Ітератор на перший елемент.
end()	Ітератор на позицію після останнього елемента.
rbegin()	Зворотний ітератор на останній елемент.
rend()	Зворотний ітератор на позицію перед першим елементом.
cbegin	<code>const_iterator</code> на початок вектора.
cend	<code>const_iterator</code> на кінець вектора.
crbegin	<code>const_reverse_iterator</code> на зворотний початок.
crend	<code>const_reverse_iterator</code> на зворотний кінець.
insert()	Вставляє елементи у вказану позицію.
erase()	Видаляє елементи за позицією або діапазоном.

Методи

swap()	Обмінює вміст з іншим вектором.
clear()	Ітератор на позицію після останнього елемента.
emplace()	Створює і вставляє елемент у вказаній позиції.
emplace_back()	Створює і вставляє елемент у кінець вектора.
assign()	Замінює елементи новими значеннями.
capacity()	Розмір зарезервованої пам'яті (у кількості елементів).
reserve()	Резервує пам'ять для заданої кількості елементів.
shrink_to_fit()	Оптимізує пам'ять, звільняючи невикористане місце.
data()	Вказівник на внутрішній масив елементів.
get_allocator()	Повертає алокатор, пов'язаний із вектором.

Швидкодія класу вектор

Операція	Часова складність
Вставка елемента в кінець	O(1) якщо достатньо capacity O(n) коли не вистачає capacity
Вставка елемента всередину (в довільне місце)	O(n)
Видалення елемента з кінця	O(1)
Видалення елемента зсередини	O(n)
Доступ до елемента за індексом	O(1)
Перегляд (прохід) по вектору	O(n)
Пошук елемента за значенням	O(n)

Приклад (Створення вектору)

```
#include <bits/stdc++.h>

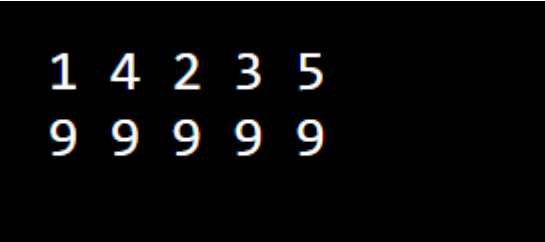
using namespace std;

void printVector(vector<int>& v) {
    for (auto x: v) {
        cout << x << " ";
    }
    cout << endl;
}

int main() {
    // Creating a vector of 5 elements from
    // initializer list
    vector<int> v1 = {1, 4, 2, 3, 5};

    // Creating a vector of 5 elements with
    // default value
    vector<int> v2(5, 9);

    printVector(v1);
    printVector(v2);
    return 0;
}
```



1 4 2 3 5
9 9 9 9 9

Приклад (Обхід вектору)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

    // Traversing vector
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";

    return 0;
}
```



a c f d z

Приклад (Оновлення елементів вектору)

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

    // Updating values using indexes 3 and 2
    v[3] = 'D';
    v.at(2) = 'F';

    cout << v[3] << endl;
    cout << v.at(2);
    return 0;
}
```



D
F

Приклад (Додавання елемента в кінець)

Коли `size()` перевищує `capacity()`, пам'ять перевиділяється, і вартість стає $O(n)$.

Метод `shrink_to_fit()` в C++ дозволяє **зменшити місткість** вектора до його фактичного розміру

```
int main() {
    std::vector<int> vec;
    for (int i = 0; i < 30; i++) {
        vec.push_back(i);
        std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;
        vec.shrink_to_fit();
    }
    vec.shrink_to_fit(); // Звільнити непотрібну пам'ять
    std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;

    return 0;
}
```

```
Size: 1, Capacity: 1
Size: 2, Capacity: 2
Size: 3, Capacity: 3
Size: 4, Capacity: 4
Size: 5, Capacity: 6
Size: 6, Capacity: 7
Size: 7, Capacity: 9
Size: 8, Capacity: 10
Size: 9, Capacity: 12
Size: 10, Capacity: 13
Size: 11, Capacity: 15
Size: 12, Capacity: 16
Size: 13, Capacity: 18
Size: 14, Capacity: 19
Size: 15, Capacity: 21
Size: 16, Capacity: 22
Size: 17, Capacity: 24
Size: 18, Capacity: 25
Size: 19, Capacity: 27
Size: 20, Capacity: 28
Size: 21, Capacity: 30
Size: 22, Capacity: 31
Size: 23, Capacity: 33
Size: 24, Capacity: 34
Size: 25, Capacity: 36
Size: 26, Capacity: 37
Size: 27, Capacity: 39
Size: 28, Capacity: 40
Size: 29, Capacity: 42
Size: 30, Capacity: 43
Size: 30, Capacity: 30
```

Приклад (Використання `reserve()`)

Метод `reserve()` лише **резервує пам'ять** для певної кількості елементів, не змінюючи фактичну кількість елементів у векторі.

```
int main() {  
    const size_t N = 30;  
  
    std::vector<int> vec;  
  
    vec.reserve(N); // Резервуємо пам'ять для 30 елементів  
  
    for (int i = 0; i < N; i++) {  
        vec.push_back(i);  
        std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;  
    }  
    return 0;  
}
```

```
Size: 2, Capacity: 30  
Size: 3, Capacity: 30  
Size: 4, Capacity: 30  
Size: 5, Capacity: 30  
Size: 6, Capacity: 30  
Size: 7, Capacity: 30  
Size: 8, Capacity: 30  
Size: 9, Capacity: 30  
Size: 10, Capacity: 30  
Size: 11, Capacity: 30  
Size: 12, Capacity: 30  
Size: 13, Capacity: 30  
Size: 14, Capacity: 30  
Size: 15, Capacity: 30  
Size: 16, Capacity: 30  
Size: 17, Capacity: 30  
Size: 18, Capacity: 30  
Size: 19, Capacity: 30  
Size: 20, Capacity: 30  
Size: 21, Capacity: 30  
Size: 22, Capacity: 30  
Size: 23, Capacity: 30  
Size: 24, Capacity: 30  
Size: 25, Capacity: 30  
Size: 26, Capacity: 30  
Size: 27, Capacity: 30  
Size: 28, Capacity: 30  
Size: 29, Capacity: 30  
Size: 30, Capacity: 30
```

Приклад (Створення вектора заданого розміру)

```
std::vector<T> vec(n); // Створення вектора з n елементів типу T
```

Для кожного елементу типу `T` буде викликаний конструктор за замовчуванням. Слід зауважити що `reserve()` не ініціалізує елементи вектора, але лише резервує пам'ять (тобто конструктор за замовчуванням не викликається в цьому випадку).

```
class MyClass {
public:
    int value;
    MyClass() : value(-1) {} // Конструктор за замовчуванням
};

int main() {
    std::vector<MyClass> vec(30); // Створення вектора з 30 елементів

    std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;

    // Виведемо значення кожного елемента
    for (const auto& obj : vec) {
        std::cout << obj.value << " ";
    }

    return 0;
}
```

```
Size: 30, Capacity: 30
```

[illegible]

Приклад (Використання `resize(n)`)

`resize(n)` – змінює розмір вектора

- ❑ Якщо `n > size()`, то додає нові елементи (викликає конструктор за замовч. `T()` для типу `T`).
- ❑ Якщо `n < size()`, то видаляє зайві елементи.
- ❑ Змінює `size()`, але не гарантує зміну `capacity()`.

```
int main() {  
    std::vector<int> vec = { 1, 2, 3 };  
    vec.resize(30); // Додає елементи (0 за замовчуванням)  
  
    std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() << std::endl;  
    for (int num : vec) std::cout << num << " ";  
  
    return 0;  
}
```

```
Size: 30, Capacity: 30
```

```
1 2 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Тестовий приклад

Яким буде результат виконання програми?

```
void main()
{
    const size_t N = 10000;
    std::vector<int> vect{ 0 };
    auto it = vect.begin();
    std::cout << *it << std::endl;
    for (int i = 1; i < N; i++)
    {
        vect.push_back(i);
        std::cout << *(it + i) << std::endl;
    }
}
```

Динамічне виділення пам'яті

- ❑ **std::vector** зберігає елементи у **неперервному блоці пам'яті**, який виділяється динамічно з купи (heap). Початково пам'ять може не виділятися (до першого додавання елементів). При додаванні елементів (через **push_back**, **emplace_back**, **insert** тощо), вектор **може перевиділяти пам'ять**, якщо поточного обсягу (**capacity**) не вистачає.
- ❑ **size()** — скільки елементів фактично збережено.
- ❑ **capacity()** — скільки елементів вміщається без перерозподілу пам'яті.
- ❑ Якщо кількість елементів відома наперед бажано явно резервувати пам'ять:

```
std::vector<int> v;  
v.reserve(100); // зарезервує місце під 100 елементів, але size все ще 0
```


Автоматичний перерозподіл пам'яті

Якщо `size() == capacity()`, наступне додавання елементу викличе **перерозподіл пам'яті**:

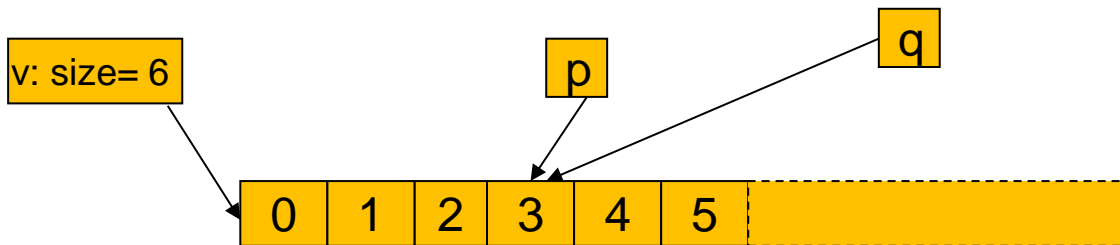
- Виділяє нову область пам'яті.
- Копіює (або переміщує) елементи зі старої області.
- Звільняє стару пам'ять.

capacity збільшується в **1.5-2 рази** при кожному виділенні (залежить від реалізації STL), це зменшує кількість виділень пам'яті та забезпечує амортизовану $O(1)$ складність при додаванні елементів в кінець вектору

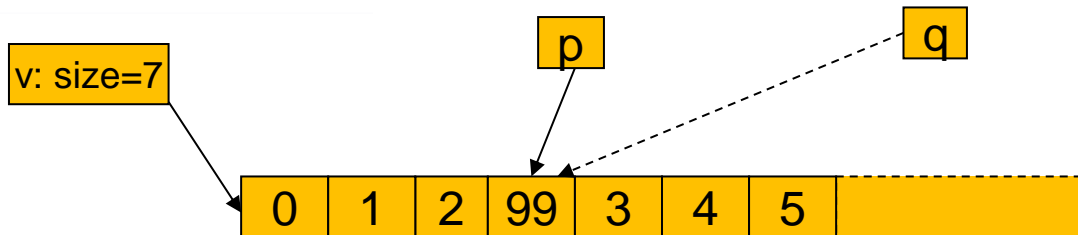
shrink-to-fit - Після видалення багатьох елементів **capacity** може залишатись великою. Можна звільнити зайву пам'ять:

```
v.shrink_to_fit(); // не гарантується, але зазвичай зменшує capacity до size
```

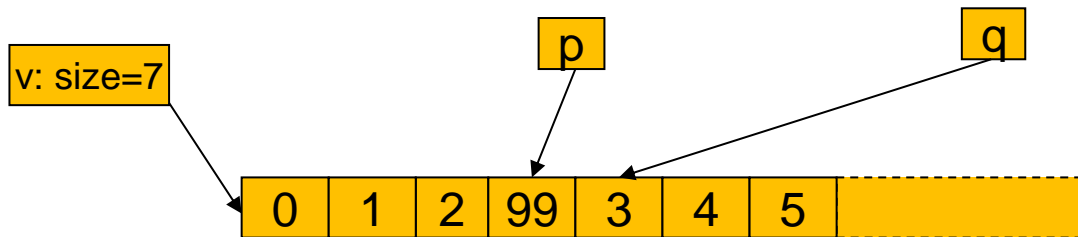
Insert()



```
int main() {  
    std::vector<int> v = { 0, 1, 2, 3, 4, 5};  
  
    auto p = v.begin(); //std::vector<int>::iterator p = v.begin();  
    p = p + 3;  
    auto q = p;  
    std::cout << *q << '\n';  
    p = v.insert(p, 99); // p указывает на вставленный элемент  
    for (int x : v) {  
        std::cout << x << " ";  
    }  
    // std::cout << *q << '\n'; //invalid iterator  
}
```

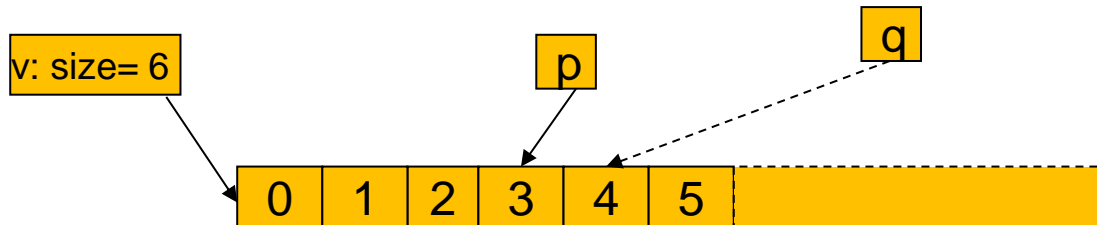


Erase()



```
q = p + 1;  
p = v.erase(p); // p вказує на елемент що слідує після видаленого  
std::cout << *p << '\n';  
for (int x : v) {  
    std::cout << x << " ";  
}  
std::cout << '\n';  
//std::cout << *q << '\n'; //invalid iterator
```

- Елементи вектора рухаються (зміщуються) після **insert()** або **erase()**
- Ітератори вектора стають недійсними після **insert()** і **erase()**



Дякую