

# Лекція 26.

## Класифікація алгоритмів STL.

### Алгоритми, які не модифікують послідовність. Лямбда вирази



# План на сьогодні

1

Класифікація алгоритмів STL. Основні принципи реалізації.

2

Приклади використання алгоритмів, які не модифікують послідовність.

3

Функтори, об'єкт-функції та функції предикати.

4

Лямбда-вирази та їх використання в алгоритмах.

5

Функціональні адаптери STL



# Класифікація алгоритмів STL



FACULTY OF APPLIED  
MATHEMATICS AND  
INFORMATICS  
LVIV UNIVERSITY

# Алгоритми STL

- ❑ **STL алгоритми** - це набір готових функцій Standard Template Library (STL), які дозволяють ефективно працювати з колекціями даних ( vector, list, set, тощо).
- ❑ **Універсальність** - працюють з будь-яким контейнером через ітератори.
- ❑ **Шаблонність** - підтримують різні типи даних.
- ❑ **Оптимізація** - ефективна реалізація.
- ❑ Алгоритми оголошені в **<algorithm>** і **<numeric>**.
- ❑ Ми можемо поділити алгоритми на такі категорії:
  - ❑ **Немодифікуючі** алгоритми (**Non-mutating algorithms**)
  - ❑ **Модифікуючі** алгоритми (**Mutating algorithms**)

<https://en.cppreference.com/w/cpp/algorithm>

# Немодифікуючі алгоритми

- ❑ Немодифікуючі алгоритми виконують операції над послідовностями елементів, не змінюючи самі елементи. Вони здебільшого використовуються для пошуку та збору інформації про елементи в діапазоні.
- ❑ На наступному слайді наведено таблицю немодифікуючих алгоритмів у C++

Алгоритм	Опис
<b>for_each()</b>	Застосовує функцію до кожного елемента в діапазоні.
<b>find()</b>	Знаходить перший елемент, який задовольняє певну умову.
<b>find_if()</b>	Шукає перший елемент у діапазоні, який задовольняє задану умову.
<b>count()</b>	Підраховує кількість входжень заданого значення у діапазоні.
<b>count_if()</b>	Підраховує кількість елементів у діапазоні, що задовольняють умову.
<b>equal()</b>	Перевіряє, чи два діапазони рівні елемент за елементом.
<b>mismatch()</b>	Знаходить першу позицію, де два діапазони відрізняються.
<b>all_of()</b>	Перевіряє, чи всі елементи діапазону задовольняють умову.
<b>any_of()</b>	Перевіряє, чи хоча б один елемент діапазону задовольняє умову.
<b>none_of()</b>	Перевіряє, чи жоден елемент діапазону не задовольняє умову.
<b>binary_search()</b> , <b>lower_bound()</b> , <b>upper_bound()</b>	Пошук елемента в контейнері за логарифмічний час $O(\log n)$ .
<b>accumulate()</b>	Обчислює суму всіх елементів у контейнері.

# Модифікуючі алгоритми

- ❑ **Модифікуючі алгоритми** — це алгоритми, які змінюють елементи в діапазоні або змінюють порядок елементів.
- ❑ На наступному слайді наведено таблицю модифікуючих алгоритмів у C++

Алгоритм	Опис
<code>copy()</code>	Створює копію контейнера або змінної.
<code>copy_n()</code>	Копіює задану кількість елементів з одного діапазону в інший.
<code>copy_if()</code>	Копіює елементи з одного діапазону в інший, якщо вони задовольняють умову.
<code>move()</code>	Переміщує елементи з одного діапазону в інший.
<code>transform()</code>	Застосовує задану операцію до діапазону елементів і зберігає результат в іншому діапазоні.
<code>fill()</code>	Присвоює задане значення всім елементам у діапазоні.
<code>generate()</code>	Присвоює значення елементам у діапазоні шляхом виклику генератора.
<code>remove()</code>	Видаляє задані елементи з діапазону.
<code>remove_if()</code>	Видаляє елементи, що задовольняють певну умову.
<code>replace()</code>	Замінює одні елементи іншими.
<code>replace_if()</code>	Замінює елементи новими, якщо виконуються певні умови.
<code>reverse()</code>	Змінює порядок елементів у контейнері на зворотний.
<code>rotate()</code>	Повертає елементи діапазону вліво або вправо.
<code>shuffle()</code>	Перемішує елементи у випадковому порядку.
<code>sort()</code>	Сортує елементи контейнера за замовчуванням за зростанням.
<code>partial_sort()</code>	Частково сортує діапазон — тільки певну частину елементів.
<code>partition()</code>	Розділяє елементи діапазону на основі умови.
<code>merge()</code>	Об'єднує два відсортовані контейнери в один.
<code>stable_sort()</code>	Сортує елементи, зберігаючи їх відносний порядок.
<code>is_sorted()</code>	Перевіряє, чи елементи контейнера відсортовані.



# Класифікація STL алгоритмів

Клас алгоритмів	Основні приклади
Несортуючі (Non-modifying)	<code>for_each</code> , <code>find</code> , <code>count</code> , <code>all_of</code> , <code>any_of</code> , <code>none_of</code>
Модифікуючі (Modifying)	<code>copy</code> , <code>move</code> , <code>fill</code> , <code>transform</code> , <code>replace</code> , <code>remove</code> , <code>swap</code>
Сортуючі (Sorting)	<code>sort</code> , <code>stable_sort</code> , <code>partial_sort</code> , <code>nth_element</code> , <code>is_sorted</code>
Бінарний пошук (Binary search)	<code>binary_search</code> , <code>lower_bound</code> , <code>upper_bound</code> , <code>equal_range</code>
Операції над множинами (Set operations)	<code>merge</code> , <code>includes</code> , <code>set_union</code> , <code>set_intersection</code> , <code>set_difference</code> , <code>set_symmetric_difference</code>
Мінімум/Максимум (Min/Max)	<code>min</code> , <code>max</code> , <code>min_element</code> , <code>max_element</code> , <code>minmax_element</code>
Числові алгоритми (Numeric)	<code>accumulate</code> , <code>inner_product</code> , <code>partial_sum</code> , <code>adjacent_difference</code>
Інші (Special algorithms)	<code>shuffle</code> , <code>unique</code> , <code>reverse</code> , <code>rotate</code> , <code>next_permutation</code> , <code>prev_permutation</code>

# Основні принципи реалізації алгоритмів

- ❑ **Універсальність ітераторів** - Алгоритми працюють не з конкретними контейнерами (vector, list, ...), а з ітераторами, які діють як узагальнені вказівники.
- ❑ **Розділення даних і логіки** - STL чітко розділяє контейнери (де зберігаються дані) і алгоритми (які оперують над даними). Завдяки цьому код стає гнучкішим і легшим для повторного використання
- ❑ **Використання шаблонів** - Усі алгоритми реалізовані як шаблонні функції, що дає змогу працювати з будь-якими типами даних без дублювання коду.
- ❑ **Концепція діапазону [first, last)** - Майже всі STL алгоритми працюють на діапазонах, які визначаються початковим і кінцевим ітератором (first включно, last виключно).
- ❑ **Робота без копіювання** - алгоритми оперують над існуючими об'єктами через ітератори, уникаючи непотрібного копіювання даних.

# Немодифікуючі алгоритми

# for\_each

- ❑ Окрім стандартних технік циклів, таких як `for`, `while` та `do-while`, мова C++ також пропонує ще одну зручну можливість — **цикл `for_each`**, який виконує аналогічну задачу.
- ❑ Цей цикл приймає функцію, яка застосовується до кожного елемента контейнера.
- ❑ Цикл `for_each` визначений у заголовочному файлі `<algorithm>`, тому для його використання потрібно підключити цей заголовок: `#include <algorithm>`.

## Переваги `for_each`:

- ❑ Працює з будь-яким контейнером STL.
- ❑ Зменшує ймовірність помилок, які можна допустити у звичайному `for` циклі.
- ❑ Робить код більш читабельним і зрозумілим.
- ❑ Покращує загальну продуктивність коду.

# Синтаксис та реалізація `for_each`

`for_each (InputIterator start_iter, InputIterator last_iter, Function fnc)`

**start\_iter** : The **beginning** position  
from where function operations has to be executed.

**last\_iter** : The **ending** position  
till where function has to be executed.

**fnc/obj\_fnc** : The 3rd argument is a function or  
an object function which operation would be applied to each element.

```
template <typename InputIt, typename Function>
Function for_each(InputIt first, InputIt last, Function f) {
    for (; first != last; ++first) {
        f(*first);
    }
    return f; // It actually returns the function object (useful if it holds state)
}
```

# Приклад 1

// Working of for\_each loop

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

// helper function 1
void printx2(int a)
{
    cout << a * 2 << " ";
}

// helper function 2
// object type function
struct Class2
{
    void operator() (int a)
    {
        cout << a * 3 << " ";
    }
} obl;
```

```
int main()
{
    // initializing array
    int arr[5] = { 1, 5, 2, 4, 3 };
    cout << "Using Arrays:" << endl;

    // printing array using for_each using function
    cout << "Multiple of 2 of elements are : ";
    for_each(arr, arr + 5, printx2);
    cout << endl;

    // printing array using for_each using object function
    cout << "Multiple of 3 of elements are : ";
    for_each(arr, arr + 5, obl);
    cout << endl;

    // initializing vector
    vector<int> arr1 = { 4, 5, 8, 3, 1 };

    cout << "Using Vectors:" << endl;
    // printing array using for_each
    // using function
    cout << "Multiple of 2 of elements are : ";
    for_each(arr1.begin(), arr1.end(), printx2);
    cout << endl;

    // printing array using for_each
    // using object function
    cout << "Multiple of 3 of elements are : ";
    for_each(arr1.begin(), arr1.end(), obl);

    cout << endl;
}
```

```
Using Arrays:
Multiple of 2 of elements are : 2 10 4 8 6
Multiple of 3 of elements are : 3 15 6 12 9
Using Vectors:
Multiple of 2 of elements are : 8 10 16 6 2
Multiple of 3 of elements are : 12 15 24 9 3
```

# Приклад 2

```
// For_each with Exception
```

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
```

```
// Helper function 1
```

```
void printx2(int a)
{
    cout << a * 2 << " ";
    if ( a % 2 == 0)
    {
        throw a;
    }
}
```

```
// Helper function 2 object type function
```

```
struct Class2
{
    void operator() (int a)
    {
        cout << a * 3 << " ";
        if ( a % 2 == 0)
        {
            throw a;
        }
    }
} obl;
```

```
int main()
{
    // Initializing array
    int arr[5] = { 1, 5, 2, 4, 3 };
    cout << "Using Array" << endl;
    // Printing Exception using for_each using function
    try
    {
        for_each(arr, arr + 5, printx2);
    }
    catch(int i)
    {
        cout << "\nThe Exception element is : " << i ;
    }
    cout << endl;
    // Printing Exception using for_each using object function
    try
    {
        for_each(arr, arr + 5, obl);
    }
    catch(int i)
    {
        cout << "\nThe Exception element is : " << i ;
    }
    // Initializing vector
    vector<int> arr1 = { 1, 3, 6, 5, 1 };
    cout << "\nUsing Vector" << endl;
    // Printing Exception using for_each using function
    try
    {
        for_each(arr1.begin(), arr1.end(), printx2);
    }
    catch(int i)
    {
        cout << "\nThe Exception element is : " << i ;
    }
    cout << endl;
    // printing Exception using for_each using object function
    try
    {
        for_each(arr1.begin(), arr1.end(), obl);
    }
    catch(int i)
    {
        cout << "\nThe Exception element is : " << i ;
    }
}
```

Using Array

2 10 4

The Exception element is : 2

3 15 6

The Exception element is : 2

Using Vector

2 6 12

The Exception element is : 6

3 9 18

The Exception element is : 6

# Приклад 3. Сума комплексних чисел за модулем

```
class MyComplex {
private:
    double real;
    double imag;

public:
    MyComplex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    double abs() const {
        return std::sqrt(real * real + imag * imag);
    }
    double getReal() const { return real; }
    double getImag() const { return imag; }
};

// Функтор для обчислення суми модулів
struct SumModules {
    double total = 0.0;
    void operator()(const MyComplex& c) {
        total += c.abs();
    }
};
```

```
int main() {
    // Створимо вектор наших комплексних чисел
    std::vector<MyComplex> numbers = {
        MyComplex(3, 4),    // модуль = 5
        MyComplex(1, -1),   // модуль ≈ 1.4142
        MyComplex(0, 2),    // модуль = 2
        MyComplex(-5, 0)    // модуль = 5
    };

    // Обчислюємо суму модулів
    SumModules result = std::for_each(numbers.begin(), numbers.end(), SumModules{});

    std::cout << "Sum of modules = " << result.total << std::endl;

    return 0;
}
```

Sum of modules = 13.4142



# Приклад 4. Сума парних чисел вектора

```
struct SumEven {  
    int total = 0;  
  
    void operator()(int x) {  
        if (x % 2 == 0) { // Перевірка на парність  
            total += x;  
        }  
    }  
};  
  
int main() {  
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };  
  
    SumEven result = std::for_each(numbers.begin(), numbers.end(), SumEven{});  
  
    std::cout << "Sum of even numbers = " << result.total << std::endl;  
  
    return 0;  
}
```

```
Sum of even numbers = 12
```

# count

- ❑ У мові C++ `count()` — це вбудована функція, яка використовується для обчислення кількості входжень певного елемента в заданому діапазоні.
- ❑ Цей діапазон може бути будь-яким контейнером STL або масивом.

Синтаксис:

- ❑ `count(first, last, val);`

Параметри:

- ❑ **first**: Ітератор на перший елемент діапазону.
- ❑ **last**: Ітератор на елемент, що йде після останнього елемента діапазону.
- ❑ **val**: Значення, кількість входжень якого потрібно підрахувати.

Повертає:

- ❑ Якщо значення знайдено — повертає кількість його входжень.
- ❑ Якщо значення не знайдено — повертає 0.

# Реалізація count

```
template <typename InputIt, typename T>
typename std::iterator_traits<InputIt>::difference_type
count(InputIt first, InputIt last, const T& value) {
    typename std::iterator_traits<InputIt>::difference_type result = 0;
    for (; first != last; ++first) {
        if (*first == value) {
            ++result;
        }
    }
    return result;
}
```

# Приклад 5

```
int main() {  
    vector<string> v = {"Apple", "Oranges",  
                        "Apple",  
                        "Banana"};  
  
    // Count the occurrence of "Apple"  
    cout << count(v.begin(), v.end(), "Apple");  
  
    return 0;  
}
```

2

# find

- ❑ У мові C++ `find()` — це вбудована функція, яка використовується для знаходження першого входження елемента в заданому діапазоні.
- ❑ Вона працює з будь-яким контейнером, що підтримує ітератори, таким як масиви, вектори, списки тощо.

## Синтаксис:

- ❑ `find(first, last, val);`

## Параметри:

- ❑ `first`: Ітератор на перший елемент діапазону.
- ❑ `last`: Ітератор на теоретичний елемент **після** останнього елемента діапазону.
- ❑ `val`: Значення, яке потрібно знайти.

## Повертає:

- ❑ Якщо значення знайдено — повертає ітератор на його позицію.
- ❑ Якщо значення не знайдено — повертає ітератор на кінець діапазону (`last`).

# Реалізація find

```
template <typename InputIt, typename T>
InputIt find(InputIt first, InputIt last, const T& value) {
    for (; first != last; ++first) {
        if (*first == value) {
            return first; // Повертаємо ітератор на знайдений елемент
        }
    }
    return last; // Якщо не знайдено, повертаємо ітератор на кінець
}
```

# Приклад 6

```
int main() {  
    int numbers[] = { 1, 2, 3, 4, 5, 6 };  
  
    // Використовуємо вказівник на перший елемент та один після останнього  
    auto it = find(begin(numbers), end(numbers), 4);  
  
    if (it != std::end(numbers)) {  
        std::cout << "Found " << *it << " at position " << (it - begin(numbers)) << endl;  
    }  
    else {  
        cout << "Not found!" << endl;  
    }  
  
    return 0;  
}
```

Found 4 at position 3

# Бінарний пошук

- ❑ У C++ стандартна бібліотека шаблонів (STL) надає різні функції, такі як `std::binary_search()`, `std::lower_bound()`, та `std::upper_bound()`, які використовують алгоритм бінарного пошуку для різних цілей.
- ❑ Ці функції працюють **лише на відсортованих даних**.

**У STL є три основні функції для бінарного пошуку:**

- ❑ `binary_search()`
- ❑ `lower_bound()`
- ❑ `upper_bound()`



# Функції бінарного пошуку в C++ STL

## 1. `binary_search(first, last, val)`

- ❑ Перевіряє, чи існує значення `val` у відсортованому діапазоні.
- ❑ Повертає `true`, якщо знайдено, інакше — `false`.
- ❑ Часова складність:  $O(\log n)$ , де  $n$  — кількість елементів у контейнері.

## 2. `lower_bound(first, last, val)`

- ❑ Повертає ітератор на **перший елемент**, який **не менший** за `val`.
- ❑ Якщо всі елементи менші — повертає `last`.

## 3. `upper_bound(first, last, val)`

- ❑ Повертає ітератор на **перший елемент**, який **більший** за `val`.
- ❑ Якщо таких елементів немає — повертає `last`.

# Приклад 7

```
// Function for check an element whether it
// is present or not
void isPresent(vector<int> &arr, int val) {

    // using binary_search to check if val exists
    if (binary_search(arr.begin(), arr.end(), val))
        cout << val << " exists in vector";
    else
        cout << val << " does not exist";

    cout << endl;
}

int main() {
    vector<int> arr = {10, 15, 20, 25, 30, 35};

    int val1 = 15;
    int val2 = 23;

    isPresent(arr, val1);
    isPresent(arr, val2);

    return 0;
}
```

15 exists in vector  
23 does not exist

# Приклад 8

```
int main() {  
    vector<int> numbers = { 1, 2, 3, 4, 5, 6 };  
  
    int value_to_find = 4;  
  
    // Знаходимо перший елемент, який не менший за value_to_find  
    auto it = lower_bound(numbers.begin(), numbers.end(), value_to_find);  
  
    if (it != numbers.end() && *it == value_to_find) {  
        cout << "Found " << value_to_find << " at position " << (it - numbers.begin()) << endl;  
    }  
    else {  
        cout << value_to_find << " is not in the vector." << endl;  
    }  
  
    return 0;  
}
```

Found 4 at position 3

# Приклад 9

```
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };

    int value_to_find = 4;

    // Знаходимо перший елемент, який більший за value_to_find
    auto it = std::upper_bound(numbers.begin(), numbers.end(), value_to_find);

    if (it != numbers.end()) {
        std::cout << "The first element greater than " << value_to_find
                  << " is " << *it << " at position " << (it - numbers.begin()) << std::endl;
    }
    else {
        std::cout << "No element greater than " << value_to_find << std::endl;
    }

    return 0;
}
```

The first element greater than 4 is 5 at position 4

# Функції `accumulate()` та `partial_sum()` у C++ STL

- ❑ Функції `accumulate()` та `partial_sum()` використовуються для знаходження суми або іншого накопиченого значення, яке отримується шляхом додавання чи іншої бінарної операції над елементами в заданому діапазоні.
- ❑ Обидві функції є частиною **Числової бібліотеки STL (Numeric Library)** і визначені в заголовочному файлі `<numeric>`.

## `accumulate(first, last, init)`

- ❑ Обчислює **загальну суму** всіх елементів у діапазоні `[first, last)`, починаючи з початкового значення `init`.
- ❑ Можна також передати власну бінарну операцію замість додавання.

## `partial_sum(first, last, result)`

- ❑ Обчислює **накопичені суми** для кожного елемента.
- ❑ У результаті кожен елемент — це сума всіх попередніх елементів до нього включно.

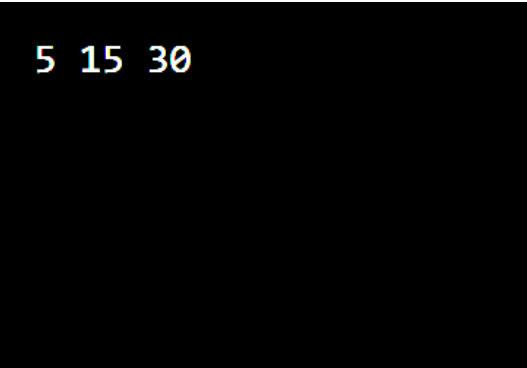
# Приклад 10

```
int main() {  
    vector<int> vec = { 5, 10, 15 };  
  
    // Defining range as whole array  
    auto first = vec.begin();  
    auto last = vec.end();  
  
    // Use accumulate to find the sum of elements in the vector  
    int sum = accumulate(first, last, 0);  
  
    cout << sum << endl;  
  
    return 0;  
}
```

30

# Приклад 11

```
int main() {  
    vector<int> vec = {5, 10, 15};  
    vector<int> res(vec.size());  
  
    // Defining range as the whole array  
    auto first = vec.begin();  
    auto last = vec.end();  
  
    // Use partial_sum to calculate the cumulative sum of elements  
    partial_sum(first, last, res.begin());  
  
    for (int val : res)  
        cout << val << " ";  
    return 0;  
}
```



5 15 30

# Функції `equal()` та `mismatch()` у C++ STL

- ❑ Функції `equal()` та `mismatch()` використовуються для порівняння елементів у діапазонах.
- ❑ Обидві функції є частиною **Стандартної бібліотеки алгоритмів STL** і визначені у заголовочному файлі `<algorithm>`.

## `equal(first1, last1, first2)`

- ❑ Порівнює елементи в діапазоні `[first1, last1)` з елементами, починаючи з `first2`.
- ❑ Повертає `true`, якщо всі відповідні елементи однакові, інакше — `false`.

## `mismatch(first1, last1, first2)`

- ❑ Знаходить першу позицію, де елементи в діапазонах `[first1, last1)` та від `first2` не збігаються.
- ❑ Повертає пару ітераторів на місця першої невідповідності.



# Приклад 12

```
int main()
{
    int v1[] = { 10, 20, 30, 40, 50 };
    std::vector<int> vector_1 (v1, v1 + sizeof(v1) / sizeof(int) );

    // Printing vector1
    std::cout << "Vector contains : ";
    for (unsigned int i = 0; i < vector_1.size(); i++)
        std::cout << " " << vector_1[i];
    std::cout << "\n";

    // using std::equal()
    // Comparison within default constructor
    if ( std::equal (vector_1.begin(), vector_1.end(), v1) )
        std::cout << "The contents of both sequences are equal.\n";
    else
        printf("The contents of both sequences differ.");
}
```

Vector contains : 10 20 30 40 50  
The contents of both sequences are equal.

# Приклад 13

```
int main() {
    std::vector<int> vec1 = { 1, 2, 3 };           // Shorter container
    std::vector<int> vec2 = { 1, 2, 3, 4, 5 };     // Longer container

    // Find the first mismatch
    auto result = std::mismatch(vec1.begin(), vec1.end(), vec2.begin());

    if (result.first != vec1.end()) {
        std::cout << "First mismatch at position: "
                  << std::distance(vec1.begin(), result.first)
                  << ", values: " << *result.first << " and " << *result.second << std::endl;
    }
    else {
        std::cout << "All elements are equal up to the end of the shorter container." << std::endl;
    }

    return 0;
}
```

All elements are equal up to the end of the shorter container.

# Функтори, об'єкт- функції та функції предикати



# Об'єкт-функція

- ❑ **Об'єкт-функція** — це будь-який об'єкт, який можна викликати як функцію.
- ❑ Має перевизначений оператор виклику `operator()`.
- ❑ Може зберігати внутрішній стан (на відміну від звичайної функції).
- ❑ Приклади: функтори, лямбди, об'єкти `std::function`.

## Об'єкт-функції

Функтори

Лямбда-функції

`std::function`

Можуть бути  
**Предикатом** (якщо  
повертає **bool**)

# Функтори

- ❑ **Функтор (об'єкт-функція)** у C++ — це **об'єкт класу** або **структури**, який можна викликати як **функцію**.
- ❑ Він перевантажує оператор виклику функції `()` і дозволяє використовувати об'єкт подібно до функції.

```
struct Adder {  
    int operator()(int a, int b) const {  
        return a + b;  
    }  
};
```

```
Adder add;
```

```
int result = add(3, 4); // працює як функція -> поверне 7
```

# Приклад 14. Функтор з внутрішнім станом

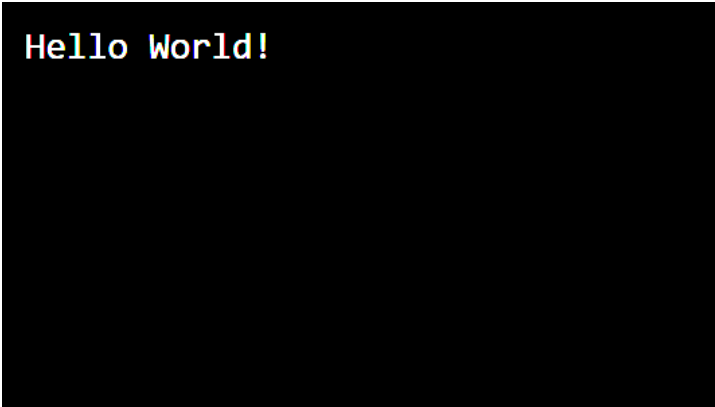
```
class AdderFixed {
    int fixed_value;
public:
    AdderFixed(int value) : fixed_value(value) {}

    int operator()(int x) const {
        return x + fixed_value;
    }
};

int main() {
    AdderFixed add5(5); // створюємо функтор, який додає 5
    std::cout << add5(10) << std::endl; // 15
    std::cout << add5(20) << std::endl; // 25
    return 0;
}
```

# Приклад 15

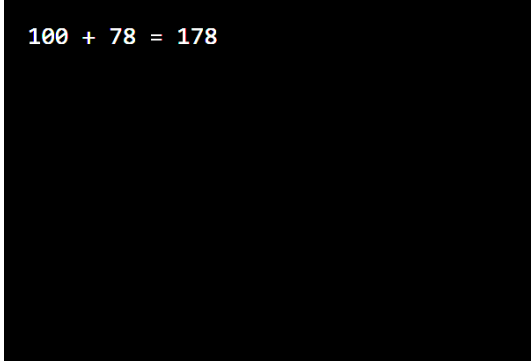
```
class Greet {  
  
    public:  
        // overload function call/parentheses operator  
        void operator() () {  
            cout << "Hello World!";  
        }  
};  
  
int main() {  
  
    // create an object of Greet class  
    Greet greet;  
  
    // call the object as a function  
    greet();  
  
    return 0;  
}
```



Hello World!

# Приклад 16

```
class Add {  
  
    public:  
        // overload function call operator  
        // accept two integer arguments  
        // return their sum  
        int operator() (int a, int b) {  
            return a + b;  
        }  
};  
  
int main() {  
  
    // create an object of Add class  
    Add add;  
  
    // call the add object  
    int sum = add(100, 78);  
  
    cout << "100 + 78 = " << sum;  
  
    return 0;  
}
```



100 + 78 = 178



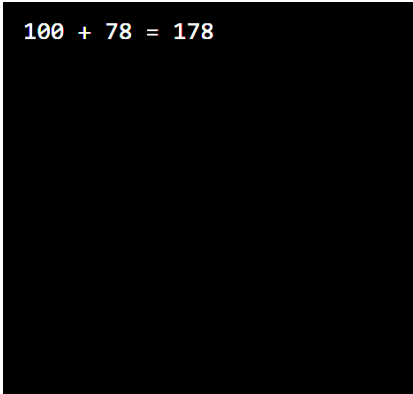
# Приклад 17 (Функтор обчислення суми)

```
class Add_To_Sum {
private:
    int initial_sum;

public:
    // constructor to initialize member variable
    Add_To_Sum(int sum) {
        initial_sum = sum;
    }
    // overload function call operator
    int operator()(int num) {
        return initial_sum += num;
    }
};

int main() {
    // create object of Add_To_Sum class
    // initialize member variable of object with value 0
    Add_To_Sum add(0);
    // call the add object with 100 as argument
    add(100);
    int final_sum = add(78);
    cout << "100 + 78 = " << final_sum;

    return 0;
}
```



100 + 78 = 178

- ❑ Функтор у C++ може містити **внутрішні змінні-члени**, які дозволяють зберігати стан між викликами або параметризувати поведінку об'єкта-функції.
- ❑ Це одна з головних переваг функтора над звичайними функціями.

# Функтори STL

- ❑ У C++ ми можемо використовувати **готові функтори**, які надає стандартна бібліотека. Для цього потрібно підключити заголовковий файл: `#include <functional>`
- ❑ C++ надає вбудовані функтори для:
  - ❑ арифметичних операцій,
  - ❑ реляційних (порівняльних) операцій,
  - ❑ логічних операцій.

# Арифметичні функтори

Functors	Description
<code>plus</code>	returns the sum of two parameters
<code>minus</code>	returns the difference of two parameters
<code>multiplies</code>	returns the product of two parameters
<code>divides</code>	returns the result after dividing two parameters
<code>modulus</code>	returns the remainder after dividing two parameters
<code>negate</code>	returns the negated value of a parameter

# Реляційні функтори (порівняльні)

Functors	Description
<code>equal_to</code>	returns <code>true</code> if the two parameters are equal
<code>not_equal_to</code>	returns <code>true</code> if the two parameters are not equal
<code>greater</code>	returns <code>true</code> if the first parameter is greater than the second
<code>greater_equal</code>	returns <code>true</code> if the first parameter is greater than or equal to the second
<code>less</code>	returns <code>true</code> if the first parameter is less than the second
<code>less_equal</code>	returns <code>true</code> if the first parameter is less than or equal to the second

# Логічні функтори

Functors	Description
<code>logical_and</code>	returns the result of Logical AND operation of two booleans
<code>logical_or</code>	returns the result of Logical OR operation of two booleans
<code>logical_not</code>	returns the result of Logical NOT operation of a boolean

# Побітові функтори

Functors	Description
<code>bit_and</code>	returns the result of Bitwise AND operation of two parameters
<code>bit_or</code>	returns the result of Bitwise OR operation of two parameters
<code>bit_xor</code>	returns the result of Bitwise XOR operation of two parameters

# Використання функторів в STL

- ❑ Функтора зазвичай використовуються разом із алгоритмами STL як аргументи до таких алгоритмів, як `sort`, `count_if`, `all_of` тощо.
- ❑ У наступному прикладі ми розглянемо використання вбудованого функтора `greater<T>()`, де `T` — це тип параметра функтора, у поєднанні з алгоритмом `sort()`.

# Приклад 18

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

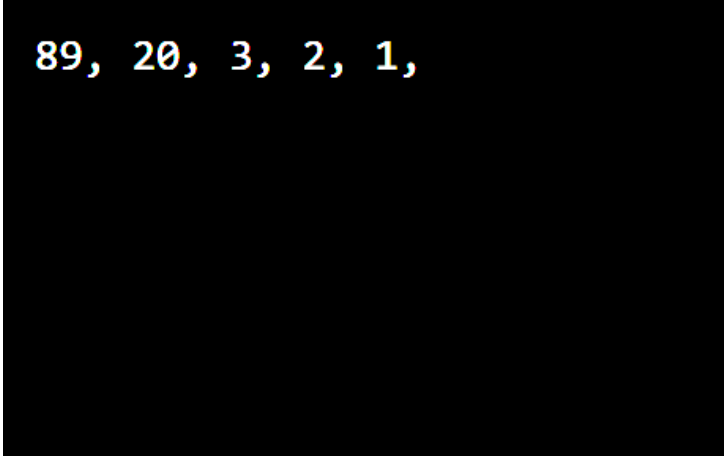
int main() {

    // initialize vector of int
    vector<int> nums = {1, 20, 3, 89, 2};

    // sort the vector in descending order
    sort(nums.begin(), nums.end(), greater<int>());

    for(auto num: nums) {
        cout << num << ", ";
    }

    return 0;
}
```



89, 20, 3, 2, 1,



# Лямбда-вирази



# Лямбда-вирази

- ❑ Лямбда-вираз у C++ дозволяє створювати **анонімні об'єкти-функції (функтори)**, які можна використовувати прямо в коді або передавати як аргументи.
- ❑ Лямбда-вирази були запроваджені у **C++11** для зручного та компактного створення анонімних функцій.
- ❑ Вони зручні, оскільки **не потрібно перерантажувати оператор `()` у окремому класі чи структурі**.

## Базовий вигляд лямбда-виразу у C++

```
auto greet = []() {  
    // тіло лямбда-функції  
};
```

Аналогічно до

```
void greet() {  
    // function body  
}
```

Тут:

- `[]` — це **лямбда-інтродуктор**, який позначає початок лямбда-виразу.
- `()` — це **список параметрів**, аналогічний до дужок у звичайній функції.

# Приклад 19

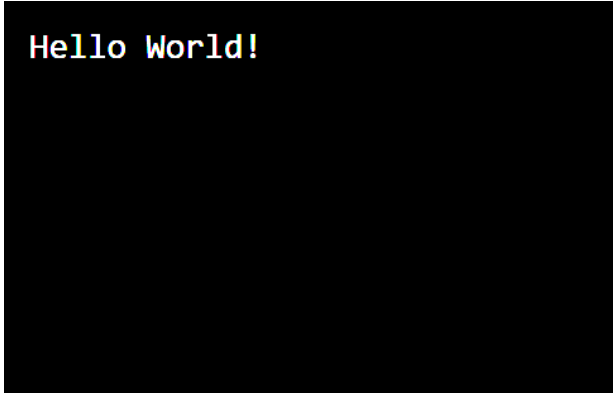
```
#include <iostream>
using namespace std;

int main() {

    // create a lambda function that prints "Hello World!"
    auto greet = []() {
        cout << "Hello World!";
    };

    // call lambda function
    greet();

    return 0;
}
```



Hello World!

# Приклад 20. Лямбда з параметрами

```
int main() {  
  
    // lambda function that takes two integer  
    // parameters and displays their sum  
    auto add = [] (int a, int b) {  
        cout << "Sum = " << a + b;  
    };  
  
    // call the lambda function  
    add(100, 78);  
  
    return 0;  
}
```

Sum = 178

# Лямбда-функції з типом повернення

Як і у звичайних функціях, лямбда-вирази у C++ також можуть мати тип повернення.

- ❑ Компілятор **може автоматично вивести тип повернення** на основі оператора `return`.

```
auto add = [] (int a, int b) {  
    // always returns an 'int'  
    return a + b;  
};
```

- ❑ Якщо ж є кілька операторів `return` з різними типами, **потрібно явно вказувати тип повернення**.

```
auto operation = [] (int a, int b, string op) -> double {  
    if (op == "sum") {  
        // returns integer value  
        return a + b;  
    }  
    else {  
        // returns double value  
        return (a + b) / 2.0;  
    }  
};
```

# Приклад 21

```
int main() {  
  
    // lambda function with explicit return type 'double'  
    // returns the sum or the average depending on operation  
    auto operation = [] (int a, int b, string op) -> double {  
        if (op == "sum") {  
            return a + b;  
        }  
        else {  
            return (a + b) / 2.0;  
        }  
    };  
  
    int num1 = 1;  
    int num2 = 2;  
  
    // find the sum of num1 and num2  
    auto sum = operation(num1, num2, "sum");  
    cout << "Sum = " << sum << endl;  
  
    // find the average of num1 and num2  
    auto avg = operation(num1, num2, "avg");  
    cout << "Average = " << avg;  
  
    return 0;  
}
```

```
Sum = 3  
Average = 1.5
```

# Захоплення змінних у лямбда-виразах C++

- ❑ **За замовчуванням** лямбда-функції не мають доступу до змінних зовнішньої функції.
- ❑ Щоб отримати доступ до них, використовується **список захоплення (capture clause)**.

## Захоплення за значенням (Capture by Value)

- ❑ Схоже на передачу аргументів у функцію за значенням.
- ❑ Копія змінної створюється під час створення лямбди.
- ❑ **Можна лише читати, але не можна змінювати** змінну всередині лямбди.

```
int num_main = 100;

// доступ до num_main за значенням
auto my_lambda = [num_main]() {
    cout << num_main;
};
```

## Захоплення за посиланням (Capture by Reference)

- ❑ Схоже на передачу аргументів у функцію за посиланням.
- ❑ Лямбда отримує доступ до **адреси змінної**.
- ❑ **Можна читати та змінювати** значення змінної.

```
int num_main = 100;

// доступ до num_main за посиланням
auto my_lambda = [&num_main]() {
    num_main = 900;
};
```

# Варіанти захоплення змінних

Захоплення	Що означає
[ = ]	Захопити всі змінні за значенням (копія)
[ & ]	Захопити всі змінні за посиланням
[ x ]	Тільки x копією
[ &x ]	Тільки x за посиланням



# Приклад 22 (захоплення за значенням)

```
int main() {  
  
    int initial_sum = 100;  
  
    // capture initial_sum by value  
    auto add_to_sum = [initial_sum] (int num) {  
        // here initial_sum = 100 from local scope  
        return initial_sum + num;  
    };  
  
    int final_sum = add_to_sum(78);  
    cout << "100 + 78 = " << final_sum;  
  
    return 0;  
}
```



100 + 78 = 178

# Приклад 22 (функтор замість лямбда виразу)

```
// Компілятор створює функтор замість лямбда виразу
class AddToSum {
    int initial_sum;

public:
    AddToSum(int sum) : initial_sum(sum) {}

    int operator()(int num) const {
        return initial_sum + num;
    }
};

int main() {
    int initial_sum = 100;

    // Використання функтора замість лямбда-виразу
    AddToSum add_to_sum(initial_sum);

    int final_sum = add_to_sum(78);
    cout << "100 + 78 = " << final_sum;

    return 0;
}
```

100 + 78 = 178

# Приклад 23 (захоплення за посиланням)

```
int main() {  
  
    int initial_sum = 0;  
  
    // capture initial_sum by reference  
    auto add_to_sum = [&initial_sum](int num) {  
        // here initial_sum = 100 from local scope  
        return initial_sum += num;  
    };  
  
    add_to_sum(100);  
    int final_sum = add_to_sum(78);  
    cout << "100 + 78 = " << final_sum;  
  
    return 0;  
}
```

100 + 78 = 178

# Приклад 23 (функтор замість лямбда виразу)

// Компілятор створює функтор замість лямбда виразу

```
class AddToSum {
    int& sum_ref;

public:
    AddToSum(int& sum) : sum_ref(sum) {}

    int operator()(int num) {
        return sum_ref += num;
    }
};

int main() {
    int initial_sum = 0;

    // Передаємо посилання до функтора
    AddToSum add_to_sum(initial_sum);

    add_to_sum(100);
    int final_sum = add_to_sum(78);
    cout << "100 + 78 = " << final_sum;

    return 0;
}
```

100 + 78 = 178

# Приклад 24 (захоплення вказівника на об'єкт поточного класу)

- ❑ Лямбда `[this](int x)` захоплює вказівник на об'єкт поточного класу
- ❑ Це дозволяє всередині лямбди звертатись до `this->base`

```
class Calculator {
private:
    int base;

public:
    Calculator(int b) : base(b) {}

    void compute_and_print(const vector<int>& nums) {
        // Лямбда захоплює this - отже, має доступ до this->base
        auto add_base = [this](int x) {
            return base += x;
        };

        for (int num : nums) {
            cout << base << " + " << num << " = " ;
            cout << add_base(num) << endl;
        }
    }
};
```

```
int main() {
    Calculator calc(100);
    vector<int> numbers = { 20, 30, 50 };
    calc.compute_and_print(numbers);

    return 0;
}
```

```
100 + 20 = 120
120 + 30 = 150
150 + 50 = 200
```

# Немодифікуючі алгоритми з лямбда виразами

Алгоритм	Що робить	Приклад лямбди
<code>find_if</code>	Знайти перший елемент	<code>[] (int x){ return x % 2 == 0; }</code>
<code>count_if</code>	Підрахувати елементи	<code>[] (int x){ return x &lt; 0; }</code>
<code>all_of</code>	Всі елементи задовольняють умову	<code>[] (int x){ return x &gt; 0; }</code>
<code>any_of</code>	Хоч один задовольняє умову	<code>[] (int x){ return x == 0; }</code>
<code>none_of</code>	Жоден не задовольняє умову	<code>[] (int x){ return x &lt; 0; }</code>
<code>for_each</code>	Виконати дію над кожним елементом	<code>[] (int x){ std::cout &lt;&lt; x; }</code>

# Приклад 25 (Обчислення кількості парних чисел)

```
int main() {  
  
    // initialize vector of integers  
    vector<int> nums = {1, 2, 3, 4, 5, 8, 10, 12};  
  
    int even_count = count_if(nums.begin(), nums.end(), [](int num) {  
        return num % 2 == 0;  
    });  
  
    cout << "There are " << even_count << " even numbers."  
  
    return 0;  
}
```

There are 5 even numbers.

## Приклад 26 (Перевірити чи всі елементи додатні)

```
int main() {
    std::vector<int> numbers = { 0, 2, 4, 6, 8 };
    bool all_positive = std::all_of(numbers.begin(), numbers.end(), [](int n) {
        return n > 0;
    });

    if (all_positive) {
        std::cout << "All numbers are positive!" << std::endl;
    }

    bool has_zero = std::any_of(numbers.begin(), numbers.end(), [](int n) {
        return n == 0;
    });

    if (has_zero) {
        std::cout << "Contains zero element!" << std::endl;
    }

    bool no_negative = std::none_of(numbers.begin(), numbers.end(), [](int n) {
        return n < 0;
    });

    if (no_negative) {
        std::cout << "No negative elements!" << std::endl;
    }
}
```

Contains zero element!  
No negative elements!



## Приклад 27 (Піднести кожен елемент до квадрату)

```
int main() {  
    std::vector<int> numbers = { 1, 2, 3, 4 };  
  
    std::for_each(numbers.begin(), numbers.end(), [](int n) {  
        std::cout << n * n << " ";  
    });  
}
```

1 4 9 16

# Функціональні адаптери STL

# Функціональні адаптери STL

Функціональні адаптери STL (або адаптери функцій) — це об'єкти або шаблони в стандартній бібліотеці C++ (STL), які дозволяють змінювати або адаптувати поведінку функцій чи функційних об'єктів. Вони використовуються, щоб зробити функціональні об'єкти сумісними з алгоритмами STL:

- ❑ `std::bind` - Адаптує функцію або метод, фіксуючи певні аргументи
- ❑ `std::function` - Універсальний контейнер для зберігання функцій, лямбд, функцій-членів, функторів
- ❑ `std::mem_fn` - Перетворює вказівник на метод класу у викликаємий об'єкт
- ❑ `std::not_fn` - Адаптер, що інвертує логіку предикатів

Для використання потрібно підключити `#include <functional>`

# std::bind

- ❑ `std::bind` - Адаптує функцію або метод, фіксуючи певні аргументи

```
int add(int a, int b) {  
    return a + b;  
}
```

```
15  
First number > 10: 12
```

```
int main() {  
    auto add10 = bind(add, 10, placeholders::_1); // перший аргумент = 10  
    cout << add10(5) << endl; // 15  
  
    vector<int> v = { 1, 5, 12, 4, 9 };  
  
    auto greater_than_10 = bind(greater<int>(), placeholders::_1, 10);  
    auto it = find_if(v.begin(), v.end(), greater_than_10);  
  
    if (it != v.end())  
        cout << "First number > 10: " << *it << endl;  
    else  
        cout << "No number > 10 found" << endl;  
}
```

# Стратегія з std::function

```
class Calculator {
private:
    std::function<double(double, double)> operation;

public:
    void setOperation(std::function<double(double, double)> op) {
        operation = op;
    }
    void compute(double a, double b) {
        if (operation)
            cout << "Result: " << operation(a, b) << endl;
        else
            cout << "Operation not set!" << endl;
    }
};

int main() {
    Calculator calc;
    calc.compute(3, 4); // Operation not set!
    // Сума
    calc.setOperation([](double a, double b) {
        return a + b;
    });
    calc.compute(10, 5); // 15
    // Добуток
    calc.setOperation([](double a, double b) {
        return a * b;
    });
    calc.compute(10, 5); // 50
    // Ділення з перевіркою
    calc.setOperation([](double a, double b) -> double {
        return (b != 0) ? a / b : 0;
    });
    calc.compute(10, 0); // 0 (захист від ділення на нуль)

    return 0;
}
```

- ❑ `std::function` - Універсальний контейнер для зберігання функцій, лямбд, функцій-членів, функторів

```
Operation not set!
Result: 15
Result: 50
Result: 0
```

# std::mem\_fn

- ❑ `std::mem_fn` - Перетворює вказівник на метод класу у викликаємий об'єкт

```
struct MyClass {  
    void say() const {  
        std::cout << "Hello!\n";  
    }  
};  
  
struct Printer {  
    void print() const {  
        cout << "Hello from Printer!" << endl;  
    }  
};  
  
int main() {  
    MyClass obj;  
    auto f = std::mem_fn(&MyClass::say);  
    f(obj); // виклик через адаптер  
  
    vector<Printer> printers(3);  
  
    // mem_fn створює адаптер для виклику методу  
    for_each(printers.begin(), printers.end(), mem_fn(&Printer::print));  
}
```

```
Hello from Printer!  
Hello from Printer!  
Hello from Printer!
```

# Дякую