Лекція 25.
Асоціативні невпорядковані контейнери.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. unordered_set
2. unordered_multiset
3. unordered_map
4. unordered_multimap
5. Підсумок

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

unordered_set

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Невпорядкована множина (unordered_set)

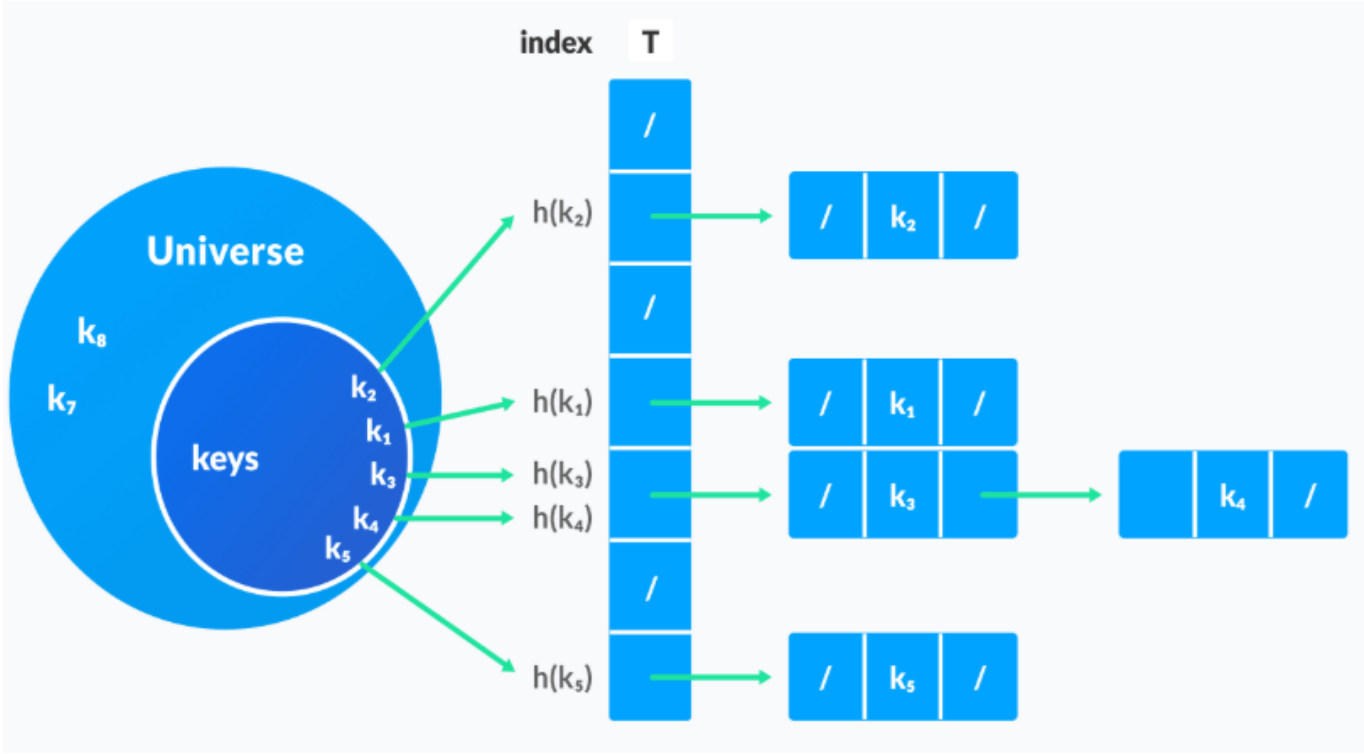
Невпорядкована множина — це контейнер з STL, який зберігає унікальні елементи в довільному порядку, використовуючи геш-таблицю. Він забезпечує швидкий доступ до елементів, але не гарантує їх порядок.

Основні особливості unordered_set:

- Унікальні значення — дублікати автоматично ігноруються
- Швидкий пошук, вставка і видалення елементів (в середньому O(1))
- Використовує геш-функції, тому ефективність залежить від їх якості
- Не підтримує доступ до елементів за індексом
- Підтримка стандартних методів STL (insert(), find(), erase(), count() тощо)

Для того щоб використовувати невпорядковану множину, потрібно підключити
#include <unordered_set>

Приклад реалізації геш-таблиці



Основні методи

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

`bool empty() const`	Перевіряє, чи контейнер порожній
`size_type size() const`	Повертає кількість елементів
`void clear()`	Видаляє всі елементи
`size_type count(const Key& key) const`	Повертає 1, якщо ключ існує, інакше 0
`iterator find(const Key& key)`	Повертає ітератор на елемент або end()
`pair<iterator, bool> insert(const Key& key)`	Додає елемент, якщо він відсутній
`iterator erase(const_iterator pos)`	Видаляє елемент за ітератором
`size_type erase(const Key& key)`	Видаляє елемент за значенням ключа
`void swap(unordered_set& other)`	Обмінює вміст із іншим unordered_set

Основні методи

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

`iterator begin()`	Ітератор на початок
`iterator end()`	Ітератор на кінець
`const_iterator cbegin()`	Константний початок
`const_iterator cend()`	Константний кінець
`void merge(unordered_set& source)`	Переміщує елементи з іншої множини
`node_type extract(const_iterator pos)`	Виймає елемент за ітератором
`node_type extract(const Key& key)`	Виймає елемент за ключем
`unordered_set& operator=(...)`	Присвоєння

Основні методи

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

`iterator begin()`	Ітератор на початок
`iterator end()`	Ітератор на кінець
`const_iterator cbegin()`	Константний початок
`const_iterator cend()`	Константний кінець
`void merge(unordered_set& source)`	Переміщує елементи з іншої множини
`node_type extract(const_iterator pos)`	Виймає елемент за ітератором
`node_type extract(const Key& key)`	Виймає елемент за ключем
`unordered_set& operator=(...)`	Присвоєння

Основні методи

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

load_factor()	коефіцієнт заповнення = середня кількість елементів в одній корзині (load_factor = size() / bucket_count()).
float max_load_factor() const	Повертає поточний максимальний коефіцієнт заповнення (за замовчуванням = 1). Це значення, яке встановлюється для обмеження середньої кількості елементів, які можуть бути розміщені в одній корзині контейнера.
void max_load_factor(float ml)	Встановлення нового значення змінює максимальну кількість елементів, які можуть бути розміщені в одній корзині перед тим, як буде викликана операція перебудови (rehash) контейнера.
size_type bucket_count() const	Кількість корзин в контейнері.

Основні методи

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

reserve(count)	Встановлює кількість "корзин" (buckets) такою, яка необхідна для розміщення щонайменше *count* елементів без перевищення максимального коефіцієнта заповнення (maximum load factor). Фактично викликає rehash(std::ceil(count / max_load_factor())).
rehash(count)	Змінює кількість "корзин" (buckets) на значення *n*, яке не менше за count і задовольняє умову *n* >= size() / max_load_factor(), після чого перебудовує контейнер, тобто розміщує елементи у відповідні корзини з урахуванням того, що загальна кількість корзин змінилася.

Швидкодія

Опис	Складність
Додавання елемента (insert())	O(1) *(в середньому)*
Видалення елемента (erase())	O(1) *(в середньому)*
Пошук елемента (find(), count())	O(1) *(в середньому)*
Доступ за ітератором (begin(), end())	O(1)
Перевірка на наявність (contains())	O(1) *(з C++20)*
Обмін вмісту (swap())	O(1)
Очищення (clear())	O(n)
Перевірка порожнечі (empty())	O(1)
Отримання розміру (size())	O(1)
Перерахунок кошиків (rehash())	O(n)

load_factor

```
unordered_set<int> sample;
cout << "The size is: " << sample.size();
cout << "\nThe bucket_count is: " << sample.bucket_count();
cout << "\nThe load_factor is: " << sample.load_factor();
cout << "\nThe max_load_factor is: " << sample.max_load_factor();
sample.insert(1);
sample.insert(11);
sample.insert(111);
sample.insert(12);
sample.insert(13);
cout << "\n\nThe size is: " << sample.size();
cout << "\nThe bucket_count is: " << sample.bucket_count();
cout << "\nThe load_factor is: " << sample.load_factor();
sample.insert(2);
sample.insert(22);
sample.insert(33);
cout << "\n\nThe size is: " << sample.size();
cout << "\nThe bucket_count is: " << sample.bucket_count();
cout << "\nThe load_factor is: " << sample.load_factor();

sample.insert(34);
cout << "\n\nThe size is: " << sample.size();
cout << "\nThe bucket_count is: " << sample.bucket_count();
cout << "\nThe load_factor is: " << sample.load_factor();
```

```
The size is: 0
The bucket_count is: 8
The load_factor is: 0
The max_load_factor is: 1
```

```
The size is: 5
The bucket_count is: 8
The load_factor is: 0.625
```

```
The size is: 8
The bucket_count is: 8
The load_factor is: 1
```

```
The size is: 9
The bucket_count is: 64
The load_factor is: 0.140625
```

Automated rehash and reserve

```
int main() {
    const int N = 10000000; // 10 millions elements

    // Without reserve
    std::unordered_set<int> s1;
    auto start1 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        s1.insert(i); // Insert elements without reserving memory
    }
    auto end1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration1 = end1 - start1;

    // With reserve
    std::unordered_set<int> s2;
    s2.reserve(N); // Pre-allocate memory for N elements
    auto start2 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        s2.insert(i); // Insert elements after reserving memory
    }
    auto end2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration2 = end2 - start2;

    // Output the elapsed time for both cases
    std::cout << "Without reserve: " << duration1.count() << " seconds\n";
    std::cout << "With reserve: " << duration2.count() << " seconds\n";

    return 0;
}
```

```
Without reserve: 10.1848 seconds
With reserve: 5.64561 seconds
```

Додавання та перевірка наявності

Метод insert() додає елемент, якщо його ще немає. Метод count() або find() дозволяє перевірити, чи елемент уже міститься в множині.

```
int main() {
    unordered_set<int> numbers;
    numbers.insert(10);
    numbers.insert(5);
    numbers.insert(10); // не буде додано вдруге
    if (numbers.count(5)) {
        cout << "Є 5" << endl;
    }
    for (int x : numbers) {
        cout << x << " ";
    }
    // Можливий Результат: 5 10
}
```

Видалення елементів

Метод erase() дозволяє видалити елемент за значенням або за ітератором. Якщо елемент не існує — нічого не відбувається.

```
int main() {
    unordered_set<int> numbers;
    numbers.insert(10);
    numbers.insert(5);
    numbers.insert(20);
    numbers.erase(5);       // видаляє елемент 5
    numbers.erase(100);     // нічого не робить
    for (int x : numbers) {
        cout << x << " ";
    }
    // Можливий Результат: 10 20
}
```

Пошук елемента

Метод find() повертає ітератор на елемент, якщо він існує, або end(), якщо ні. Можна використати для доступу або перевірки.

```
int main() {
    unordered_set<int> numbers;
    numbers.insert(1);
    numbers.insert(2);
    numbers.insert(3);
    unordered_set<int>::iterator it = numbers.find(2);
    if (it != numbers.end()) {
        cout << "Знайдено: " << *it << endl;
    } else {
        cout << "Не знайдено" << endl;
    }
}
```

Очищення множини

Метод clear() повністю видаляє всі елементи з unordered_set, роблячи його порожнім.

```
int main() {
    unordered_set<int> numbers;
    numbers.insert(7);
    numbers.insert(8);
    numbers.insert(9);
    numbers.clear();
    if (numbers.empty()) {
        cout << "Множина порожня" << endl;
    }
}
```

Об’єднання множин

Метод merge() переносить елементи з однієї unordered_set до іншої. Дублікати не додаються.

```
int main() {
    unordered_set<int> a = {1, 2};
    unordered_set<int> b = {2, 3};
    a.merge(b);
    for (int x : a) {
        cout << "a: " << x << endl;
    }
    for (int x : b) {
        cout << "b: " << x << endl;
    }
    // Можливий Результат:
    // a: 1, 2, 3
    // b: 2
}
```

Чому "Можливий результат"?

це невпорядкований контейнер, тобто:

- він не гарантує порядок зберігання елементів;
- навіть якщо додати значення в певному порядку, реальний порядок обходу (через for (x : container)) може змінюватися залежно від:
  - хеш-функції,
  - внутрішньої реалізації контейнера,
  - компілятора або навіть версії STL.

тому:

```
unordered_multiset<int> numbers = {5, 5, 10, 10};
    for (int x : numbers) cout << x << " ";
```

може вивести:

- `5 5 10 10`
- `10 5 10 5`
- `5 10 5 10`
- ...або будь-яку перестановку!

Додавання елементів типу Point в unordered_set

Задача: зберегти унікальні точки типу Point в unordered_set.

Щоб зберегти об’єкти власного типу в unordered_set потрібно:

- ❏ Задати власну геш-функцію для класу (hash(key) - визначається номер бакета)
- ❏ Перевантажити оператор порівняння або визначити функцію порівняння (== використовується щоб перевірити чи такого елемента ще немає)

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

Додавання елементів типу Point в unordered_set

```cpp
struct Point {
    int x, y;

    //// Define equality operator to compare two points
    //bool operator==(const Point& other) const {
    //    return x == other.x && y == other.y;
};

// Custom hash function for Point
struct PointHasher {
    std::size_t operator()(const Point& p) const {
        return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
    }
};

// Custom equality function for Point
struct PointEqual {
    bool operator()(const Point& a, const Point& b) const {
        // Compare two points for equality
        return (a.x == b.x) && (a.y == b.y);
    }
};
```

```cpp
int main() {
    // Declare unordered_set using custom hash and custom equality
    std::unordered_set<Point, PointHasher, PointEqual> points;

    // Insert some points
    points.insert({ 1, 2 });
    points.insert({ 3, 4 });
    points.insert({ 1, 2 }); // Duplicate - will not be added

    // Output the points
    for (const auto& p : points) {
        std::cout << "(" << p.x << ", " << p.y << ")\n";
    }

    return 0;
}
```

```
(1, 2)
(3, 4)
```

unordered_multiset

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Невпорядкована мультимножина (unordered_multiset)

Невпорядкована мультимножина — це контейнер з STL, який дозволяє зберігати кілька однакових значень у довільному порядку. Контейнер реалізовано на основі геш-таблиці, тому він забезпечує швидкий доступ до елементів, але не гарантує порядок.

Основні особливості unordered_multiset:

- Дозволяє дублікати — однакові значення зберігаються кілька разів
- Швидке додавання, пошук і видалення елементів (в середньому O(1))
- Використовує хеш-функції, як і unordered_set
- Не підтримує доступ за індексом
- Підтримка стандартних методів STL (insert(), find(), erase(), equal_range(), count() тощо)

Для того щоб використовувати невпорядковану мультимножину, потрібно підключити
#include <unordered_multiset>

Основні методи

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

`iterator insert(const Key& key)`	Додає елемент (дозволяє дублікати)
`iterator erase(const_iterator pos)`	Видаляє один екземпляр елемента
`size_type erase(const Key& key)`	Видаляє **всі** елементи з таким значенням

Інші методи (empty, size, clear, find, count, swap, load_factor) — однакові з unordered_set

Додавання дублікатів

На відміну від unordered_set, unordered_multiset дозволяє додавати однакові значення кілька разів.

```
int main() {
    unordered_multiset<int> numbers;
    numbers.insert(5);
    numbers.insert(10);
    numbers.insert(5);
    numbers.insert(10);
    for (int number : numbers) {
        cout << number << " ";
    }
    // Можливий результат: 5 5 10 10
}
```

Отримання всіх копій значення

Метод equal_range() повертає пару ітераторів на всі копії елемента з певним значенням у unordered_multiset.

```
int main() {
    unordered_multiset<int> numbers = {10, 20, 10, 30, 10};
    pair<unordered_multiset<int>::iterator, unordered_multiset<int>::iterator> range =
         numbers.equal_range(10);

    for (unordered_multiset<int>::iterator it = range.first; it != range.second; ++it) {
        cout << *it << " ";
    }
    // Результат: 10 10 10
}
```

Видалення всіх копій значення

У unordered_multiset метод erase(key) видаляє всі екземпляри елемента з заданим значенням.

```
int main() {
    unordered_multiset<int> numbers = {5, 10, 5, 20, 5};
    numbers.erase(5);
    for (int number : numbers) {
        cout << number << " ";
    }
    // Можливий результат: 10 20
}
```

Обчислення кількості копій

Метод count() у unordered_multiset повертає кількість копій заданого значення, який на відміну від unordered_set може повертати значення > 1:

```
int main() {
    unordered_multiset<int> numbers = {7, 8, 7, 9, 7};
    cout << "Кількість елементів 7: " << numbers.count(7) << endl;
    cout << "Кількість елементів 8: " << numbers.count(8) << endl;
    // Результат:
    // Кількість елементів 7: 3
    // Кількість елементів 8: 1
}
```

unordered_map

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Невпорядкований словник (unordered_map)

Невпорядкований словник — це асоціативний контейнер з STL, який зберігає пари ключ–значення у довільному порядку. Оснований на геш-таблиці, тому забезпечує швидкий доступ за ключем, але не гарантує порядок зберігання.

Основні особливості unordered_map:

- Зберігає пари ключ–значення
- Швидкий доступ, вставка і видалення за ключем (у середньому O(1))
- Кожен ключ унікальний, значення можуть повторюватися
- Доступ до значення за ключем через operator[] або at()
- Підтримка методів STL (insert(), find(), erase(), count(), operator[], at() тощо)

Для того щоб використовувати невпорядкований словник, потрібно підключити
#include <unordered_map>

Основні методи

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

`bool empty() const`	Перевіряє, чи контейнер порожній
`size_type size() const`	Повертає кількість елементів
`void clear()`	Видаляє всі елементи
`size_type count(const Key& key) const`	Повертає 1, якщо ключ існує, інакше 0
`iterator find(const Key& key)`	Повертає ітератор на елемент або end()
`pair<iterator, bool> insert(const value_type& val)`	Додає пару, якщо ключу ще немає
`iterator erase(const_iterator pos)`	Видаляє елемент за ітератором
`size_type erase(const Key& key)`	Видаляє елемент за ключем
`void swap(unordered_map& other)`	Обмінює вміст із іншим unordered_map
`float load_factor() const`	Середня кількість елементів на кошик

Основні методи

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

`mapped_type& operator[](const Key& key)`	Доступ до значення за ключем (додає, якщо відсутній)
`mapped_type& at(const Key& key)`	Доступ до значення з перевіркою існування
`const_iterator begin() const`	Початок контейнера (для читання)
`const_iterator end() const`	Кінець контейнера (для читання)
`iterator begin()`	Початок контейнера
`iterator end()`	Кінець контейнера
`template<Args...> emplace(Args&&...)`	Додає новий елемент без копіювання
`template<Args...> try_emplace(...)`	Вставляє, якщо ключ відсутній (з C++17)
`hasher hash_function() const`	Повертає хеш-функцію
`key_equal key_eq() const`	Повертає об'єкт для порівняння ключів

Швидкодія

Опис	Складність
Додавання пари (insert(), emplace())	O(1) *(в середньому)*
Доступ до значення (operator[], at())	O(1) *(в середньому)*
Видалення елемента за ключем (erase(key))	O(1) *(в середньому)*
Пошук за ключем (find(), count())	O(1) *(в середньому)*
Доступ до елементів через ітерацію	O(n)
Перевірка наявності (count())	O(1)
Очищення (clear())	O(n)
Перевірка порожнечі (empty())	O(1)
Визначення розміру (size())	O(1)
Обмін вмістом (swap())	O(1)

Додавання пар ключ–значення

Метод operator[] додає нову пару, якщо ключ ще не існує, або змінює значення, якщо такий ключ вже є.

```
int main() {
    unordered_map<string, int> ages;
    ages["Anna"] = 25;
    ages["Bohdan"] = 30;
    ages["Anna"] = 28; // оновлення значення
    for (pair<string, int> entry : ages) {
        cout << entry.first << ": " << entry.second << endl;
    }
    // Можливий результат:
    // Anna: 28
    // Bohdan: 30
}
```

Доступ до значення з перевіркою

Метод at() повертає значення за ключем, але викидає виняток, якщо ключ відсутній.

```
int main() {
    unordered_map<string, int> scores;
    scores["Alice"] = 90;
    scores["Bob"] = 85;
    cout << "Оцінка Alice: " << scores.at("Alice") << endl;
    // cout << scores.at("Eve"); // викличе виняток, бо ключ відсутній
}
```

Додавання елементів через insert()

Метод insert() додає нову пару, тільки якщо ключ ще не існує. Інакше нічого не змінює.

```
int main() {
    unordered_map<string, int> salaries;
    salaries.insert({"Ihor", 5000});
    salaries.insert({"Ihor", 7000}); // не буде оновлено
    for (pair<string, int> person : salaries) {
        cout << person.first << ": " << person.second << endl;
    }
    // Результат:
    // Ihor: 5000
}
```

Пошук значення за ключем

Метод find() повертає ітератор на елемент із заданим ключем або end(), якщо його немає.

```
int main() {
    unordered_map<string, int> stock = {
        {"apple", 50},
        {"banana", 30}
    };
    unordered_map<string, int>::iterator it = stock.find("apple");
    if (it != stock.end()) {
        cout << "Є " << it->first << ": " << it->second << " шт." << endl;
    } else {
        cout << "Немає такого товару" << endl;
    }
    // Результат:
    // Є apple: 50 шт.
}
```

Видалення елемента за ключем

Метод erase() видаляє пару з unordered_map, якщо ключ існує.

```
int main() {
    unordered_map<string, int> ages = {
        {"Ivan", 40},
        {"Oleh", 35}
    };
    ages.erase("Oleh");
    for (pair<string, int> person : ages) {
        cout << person.first << ": " << person.second << endl;
    }
    // Результат:
    // Ivan: 40
}
```

Видалення елемента за ключем

Метод erase() видаляє пару з unordered_map, якщо ключ існує.

```
int main() {
    unordered_map<string, int> ages = {
        {"Ivan", 40},
        {"Oleh", 35}
    };
    ages.erase("Oleh");
    for (pair<string, int> person : ages) {
        cout << person.first << ": " << person.second << endl;
    }
    // Результат:
    // Ivan: 40
}
```

Обчислення кількості унікальних елементів

Знайти кількість входжень кожної точки у заданий unordered_multiset<Point>.

```cpp
struct Point {
    int x, y;

    // Equality operator needed for unordered_multiset
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
};

// Define a custom hash function for Point
struct PointHasher {
    std::size_t operator()(const Point& p) const {
        return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
    }
};
```

```
Point counts:
(1, 2) appears 3 times
(3, 4) appears 2 times
(5, 6) appears 1 times
```

```cpp
int main() {
    // Define an unordered_multiset with custom hash and equality
    std::unordered_multiset<Point, PointHasher> points;

    // Insert some Points, including duplicates
    points.insert({ 1, 2 });
    points.insert({ 3, 4 });
    points.insert({ 1, 2 });
    points.insert({ 5, 6 });
    points.insert({ 1, 2 });
    points.insert({ 3, 4 });

    // Count how many times each unique point appears
    std::unordered_map<Point, int, PointHasher> point_counts;

    for (const auto& p : points) {
        point_counts[p]++;
    }

    // Print the counts
    std::cout << "Point counts:\n";
    for (const auto& entry : point_counts) {
        std::cout << "(" << entry.first.x << ", " << entry.first.y << ")"
            << " appears " << entry.second << " times\n";
    }

    return 0;
}
```

unordered_multimap

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Невпорядкований мультисловник (unordered_multimap)

Невпорядкований мультисловник — це контейнер з STL, який дозволяє зберігати кілька пар з однаковими ключами у довільному порядку. Контейнер реалізований на основі хеш-таблиці, тому забезпечує швидкий доступ за ключем, але не гарантує порядок.

Основні особливості unordered_multimap:

- Дозволяє однакові ключі — однакові ключі зберігаються кілька разів
- Швидке додавання, пошук і видалення пар (в середньому O(1))
- Кожен ключ може відповідати кільком значенням
- Не підтримує доступ через operator[]
- Підтримка методів STL (insert(), find(), erase(), equal_range(), count() тощо)

Для того щоб використовувати невпорядкований мультисловник, потрібно підключити
#include <unordered_map>

Основні методи

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

`iterator insert(const value_type& val)`	Додає пару, **навіть якщо ключ уже існує**
`size_type count(const Key& key) const`	Повертає **кількість** пар з заданим ключем
`pair<iterator, iterator> equal_range(const Key& key)`	Повертає діапазон всіх пар із цим ключем

Всі інші методи (find, erase, clear, empty, swap, load_factor тощо) — однакові з unordered_map

Додавання кількох значень для одного ключа

Контейнер unordered_multimap дозволяє додавати кілька пар з однаковим ключем.

```
int main() {
    unordered_multimap<string, int> grades;
    grades.insert({"Ivan", 80});
    grades.insert({"Ivan", 85});
    grades.insert({"Olena", 90});
    for (pair<string, int> g : grades) {
        cout << g.first << ": " << g.second << endl;
    }
    // Можливий результат:
    // Ivan: 80
    // Ivan: 85
    // Olena: 90
}
```

Отримання всіх значень одного ключа

Метод equal_range() повертає пару ітераторів, які вказують на всі пари з певним ключем.

```
int main() {
    unordered_multimap<string, int> grades = {
        {"Maria", 90},
        {"Andrii", 85},
        {"Maria", 95}
    };
    pair<unordered_multimap<string, int>::iterator, unordered_multimap<string, int>::iterator> range =
          grades.equal_range("Maria");
    for (unordered_multimap<string, int>::iterator it = range.first; it != range.second; ++it) {
        cout << it->first << ": " << it->second << endl;
    }
    // Можливий результат:
    // Maria: 90
    // Maria: 95
}
```

Обчислення кількості пар з ключем

Метод count() у unordered_multimap повертає кількість пар із заданим ключем.

```
int main() {
    unordered_multimap<string, int> prices = {
        {"book", 100},
        {"pen", 20},
        {"book", 120},
        {"book", 90}
    };
    cout << "Кількість цін для 'book': " << prices.count("book") << endl;
    cout << "Кількість цін для 'pen': " << prices.count("pen") << endl;
    // Результат:
    // Кількість цін для 'book': 3
    // Кількість цін для 'pen': 1
}
```

Видалення всіх пар з однаковим ключем

Метод erase() видаляє всі пари, які мають заданий ключ.

```
int main() {
    unordered_multimap<string, int> catalog = {
        {"lamp", 300},
        {"chair", 500},
        {"lamp", 350}
    };
    catalog.erase("lamp");
    for (pair<string, int> item : catalog) {
        cout << item.first << ": " << item.second << endl;
    }
    // Результат:
    // chair: 500
}
```

Підсумок

Загальні характеристики

Геш-контейнери забезпечують швидкий доступ до елементів за ключем, але не зберігають порядок. Вибір залежить від унікальності ключів і типу доступу.

Контейнер	Унікальні ключі	Доступ за ключем	Дублікати	Доступ за індексом	Вставка	Пошук	Видалення	Пари (ключ–значення)
unordered_set	✅	✅ O(1)*	❌	❌	✅O(1)*	✅O(1)*	✅ O(1)*	❌
unordered_multiset	❌	✅ O(1)*	✅	❌	✅O(1)*	✅O(1)*	✅ O(1)*	❌
unordered_map	✅	✅ O(1)*	❌	❌	✅O(1)*	✅O(1)*	✅ O(1)*	✅
unordered_multimap	❌	✅ O(1)*	✅	❌	✅O(1)*	✅O(1)*	✅ O(1)*	✅

* — У середньому O(1), але в найгіршому випадку — O(n) через можливі хеш-колізії.

Що коли обирати?

- unordered_set — коли потрібен швидкий доступ до унікальних значень, неважливий порядок; аналог множини з доступом O(1).
- unordered_multiset — коли треба зберігати дублікати значень із швидким доступом; множина, яка дозволяє повторення.
- unordered_map — коли потрібно зберігати пари ключ–значення з унікальними ключами; швидкий доступ до значення за ключем.
- unordered_multimap — коли потрібна асоціативна структура з дубльованими ключами, наприклад, для зберігання декількох значень на один ключ.

Всі ці структури забезпечують швидкий доступ (O(1)) у середньому випадку завдяки хеш-таблицям, але не зберігають порядок елементів.

Задача: Обчислення слів у тексті

Дано рядок з кількох слів. Потрібно обчислити, скільки разів зустрічається кожне слово.

```
int main() {
    string text = "apple banana apple orange banana apple";
    unordered_map<string, int> wordCount;
    stringstream ss(text);
    string word;
    while (ss >> word) {
        wordCount[word]++;
    }
    for (pair<string, int> entry : wordCount) {
        cout << entry.first << ": " << entry.second << endl;
    }
    // Можливий результат:
    // apple: 3
    // banana: 2
    // orange: 1
}
```

Задача: Студенти та їхні оцінки

Потрібно зберігати кілька оцінок для кожного студента, а потім вивести всі оцінки конкретного студента.

```
int main() {
    unordered_multimap<string, int> grades;
    grades.insert({"Shevchenko", 80});
    grades.insert({"Petrenko", 90});
    grades.insert({"Shevchenko", 75});
    grades.insert({"Shevchenko", 95});
    string name = "Shevchenko";
    auto range = grades.equal_range(name);
    cout << "Оцінки для " << name << ": ";
    for (auto it = range.first; it != range.second; ++it) {
        cout << it->second << " ";
    }
    // Можливий результат:
    // Оцінки для Shevchenko: 80 75 95
}
```

Задача: Перевірка унікальних логінів

У системі реєстрації потрібно перевіряти, чи логін вже використовується. Якщо логін унікальний — зареєструвати, інакше вивести повідомлення про помилку.

```
int main() {
    unordered_set<string> logins;
    string input;
    while (cin >> input && input != "exit") {
        if (logins.count(input)) {
            cout << "Логін зайнятий: " << input << endl;
        } else {
            logins.insert(input);
            cout << "Реєстрація успішна: " << input << endl;
        }
    }
    // Приклад введення: alice bob alice exit
    // Вивід:
    // Реєстрація успішна: alice
    // Реєстрація успішна: bob
    // Логін зайнятий: alice
}
```

Задача: Облік голосів

Проводиться голосування, де кожен учасник може проголосувати декілька разів за одного і того ж кандидата. Потрібно підрахувати, скільки голосів отримав кожен кандидат.

```
int main() {
    unordered_multiset<string> votes;
    votes.insert("Oksana");
    votes.insert("Petro");
    votes.insert("Oksana");
    votes.insert("Oksana");
    votes.insert("Petro");
    cout << "Голоси за Oksana: " << votes.count("Oksana") << endl;
    cout << "Голоси за Petro: " << votes.count("Petro") << endl;
    // Результат:
    // Голоси за Oksana: 3
    // Голоси за Petro: 2
}
```

Unordered_map vs map

Критерій	`std::map`	`std::unordered_map`
Внутрішня структура	Червоний-чорний дерево (Red-Black Tree)	Геш-таблиця (Hash Table)
Пошук (`find`), вставка (`insert`), видалення (`erase`)	**O(log n)**	**O(1)** (в середньому), **O(n)** (в гіршому випадку)
Порядок елементів	Відсортовані за ключем	Без порядку
Підтримка ітерацій за порядком	Легко (від найменшого до найбільшого)	Ні, порядок випадковий
Пам'ять	Менше	Більше через хеш-таблицю
Вимоги до ключа	Ключ має підтримувати оператор `<`	Ключ має бути хешованим (`std::hash`)
Підтримка діапазонних операцій (`lower_bound` , `upper_bound`)	Є	Немає
Переалокації при зростанні	Ні	Так (при зміні розміру хеш-таблиці)
Надійність часу виконання	Стабільне `O(log n)`	Середньо стабільне `O(1)` , але може бути `O(n)` при поганому хешуванні

Unordered_map vs map

```cpp
int main() {
    std::string text;
    for (int i = 0; i < 100000; ++i) {
        text += "hello world test example hello example ";
    }

    // Без оптимізації
    auto start = std::chrono::high_resolution_clock::now();
    countWordsUnorderedMap(text);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration_unordered = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Time with unordered_map (no reserve): " << duration_unordered.count() << " ms" << std::endl;

    // З оптимізацією
    start = std::chrono::high_resolution_clock::now();
    countWordsUnorderedMapOptimized(text);
    end = std::chrono::high_resolution_clock::now();
    auto duration_unordered_opt = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Time with unordered_map (reserve): " << duration_unordered_opt.count() << " ms" << std::endl;

    // map для порівняння
    start = std::chrono::high_resolution_clock::now();
    countWordsMap(text);
    end = std::chrono::high_resolution_clock::now();
    auto duration_map = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Time with map: " << duration_map.count() << " ms" << std::endl;

    return 0;
}
```

```
Time with unordered_map (no reserve): 394 ms
Time with unordered_map (reserve): 411 ms
Time with map: 459 ms
```

Рекомендації для використання

Задача	Використовувати
Частотний аналіз (підрахунок слів, символів тощо)	`unordered_map` (швидше)
Потрібна впорядкованість результатів (напр., слова за алфавітом)	`map`
Частий пошук, вставка, видалення без потреби у порядку	`unordered_map`
Пошук в діапазоні значень (`lower_bound` , `upper_bound`)	`map`
Вставка великих обсягів даних з подальшою ітерацією у порядку	`map`
Збереження невеликої кількості елементів, де ефективність не критична	будь-яку (різниця мінімальна)
Підтримка ключів зі складними типами, які важко хешувати	`map`
Робота з великим набором даних (мільйони ключів) і потрібна максимальна швидкість	`unordered_map`

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY