18. std::set, std::map, std::multi_set, std::multi_map and their member functions. Iterators.

Обзор

Контейнеры `std::set`, `std::map`, `std::multiset` и `std::multimap` входят в стандартную библиотеку C++ (заголовочные файлы `<set>` и `<map>`). Все они называются *сортированными контейнерами*. По умолчанию их реализация базируется на *красно-чёрном дереве*. Это гарантирует операции поиска, вставки и удаления за время порядка $\mathcal{O}(\log n)$.

Ниже рассматривается каждый контейнер, его ключевые особенности, основные функции-члены (member functions) и типы итераторов.

---

1. `std::set`

1.1. Назначение и свойства

- `std::set` — это *множество* уникальных элементов, упорядоченных по определённому критерию (по возрастанию с использованием `std::less<T>` по умолчанию).
- Каждый элемент в `std::set` является *ключом* и *значением* одновременно.
- Вставка *дубликатов не допускается*. Если попытаться вставить элемент, который уже содержится, вставка не произойдёт.
- Можно использовать пользовательский компаратор вместо `std::less<T>`.
- Объявление:

```
std::set<int> values;
values.insert(5);
values.insert(10);
....
values.find(-2);
values.end();
values.erase();
```

- Обход элементов двумя способами:

```
// Range-based for loop
for (int x: values) {
        std::cout << x;
}

// Using iterators
for (auto it = values.begin(); it != values.end(); ++it) {
        std::cout << *it;
}
```

1.2. Основные операции

① `insert(const value_type& val)`
- Вставляет элемент `val` в множество (если такого ещё нет).
- Возвращает пару: итератор и булево значение (удачна ли была вставка).
- Сложность: $\mathcal{O}(\log n)$.

② `erase(const value_type& key)`
- Удаляет элемент, равный `key` (если существует).
- Возвращает количество удалённых элементов (в случае `std::set` это либо `0`, либо `1`).
- Сложность: $\mathcal{O}(\log n)$.

③ `find(const value_type& key)`
- Ищет элемент, равный `key`.
- Возвращает итератор на найденный элемент(`std::set<int>::iterator`) или `end()`, если элемент не найден.
- Сложность: $\mathcal{O}(\log n)$.

④ `count(const value_type& key)`
- Возвращает количество элементов, равных `key`. Для `std::set` это либо `0`, либо `1`.
- Сложность: $\mathcal{O}(\log n)$.

⑤ `lower_bound(const value_type& key)` / `upper_bound(const value_type& key)`
- `lower_bound(key)` возвращает итератор на **первый элемент**, не меньший `key`.
- `upper_bound(key)` возвращает итератор на **первый элемент**, строго больший `key`.
- Сложность: $\mathcal{O}(\log n)$.

⑥ **Итераторы**
- `begin()` (показывает на самый левый узел, тк он самый маленький), `end()` (обычно показывает на пустой узел после наибольшего узла), `cbegin()`, `cend()`
- Итераторы `std::set` — **двунаправленные** (bidirectional).
- При вставке/удалении элементы могут быть перераспределены внутри (через операции с красно-чёрным деревом), что может инвалидировать итераторы, указывающие на изменённые (или удалённые) узлы.

1.3. Особенности

- `std::set` обычно реализуется как самобалансирующееся бинарное дерево поиска (красно-чёрное дерево).
- В отличие от последовательных контейнеров (`std::vector` и т.д.), `std::set` не может предоставить доступ по индексу, зато обеспечивает логарифмическую вставку/поиск/удаление.

---

2. `std::map`

2.1. Назначение и свойства

- `std::map` — массив, индексы которого не объязательно числа, могут быть всё что угодно.
  нпр: (индексы - строки(имена людей), а значения - int(их номер телефона)

```
std::map<std::string, int> numbers;
numbers["Alexander Hakobyan"] = 41278304;
numbers["Police"] = 102; // ("Police", 102)
// если написать
numbers["Police"] = 103; // то поменяем значение "Police" на 103 (если ключа ещё не было, добавляется новое значение)
numbers["Emergency"] = 911; // ("Emergency", 911)

// 1. Итерация с использованием итераторов
```

```cpp
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << "Key: " << it->first << ", Value: " << it->second << std::endl;
}

// 2. Итерация с использованием std::pair
for (std::pair<const std::string, int> p : numbers) {
        std::cout << "Key: " << p.first << ", Value: " << p.second << std::endl;
}

// 3. Итерация с использованием structured bindings (C++17+)
for (const auto& [key, value] : numbers) {
        std::cout << "Key: " << key << ", Value: " << value << std::endl;
}

// 4. Вставка нового элемента в map
numbers.insert(std::make_pair("Ameria", 56111111));
```

----> [Вопрос]: что меньше ("Emergency", 911) или ("Police", 102)
ответ: ("Emergency", 911), т.к. 'E' меньше.

то есть **map** - это *ассоциативный* контейнер пар вида `<ключ, значение>` (хранятся как `std::pair<const Key, T>`).

- **map** - функциональное отображение (из множества А в множество Б - подмножество декартового произведения А на Б (подмножество всех пар))
- по сути **map** - это set, просто хранятся пары `<ключ, значение>` и сравниваются по ключу.
- *Ключи уникальны*. Элементы сравниваются по ключу. Если пытаться вставить уже существующий ключ, вставка либо игнорируется, либо (в некоторых методах) обновляется значение, но второй вариант обычно относится к операциям вида `map[key] = value`.
- По умолчанию используется `std::less<Key>` для упорядочивания по ключу, можно передать кастомный компаратор.
- Можно обращаться к элементу через `operator[]` или метод `at()`, что позволяет получать/изменять `mapped_value` по ключу.
-

2.2. Основные операции

① **`operator[](const key_type& k)`**
- Возвращает **ссылку** на значение, ассоциированное с ключом `k`.
- Если ключа ещё нет в контейнере, происходит **вставка** этого ключа с созданием значения по умолчанию.
- Сложность: $\mathcal{O}(\log n)$.

② **`at(const key_type& k)`**
- Похоже на `operator[]`, но если ключ не найден, выбрасывается исключение `std::out_of_range`.
- Сложность: $\mathcal{O}(\log n)$.

③ **`insert(const value_type& val)`**
- Вставляет пару `<key, mapped_value>`. Возвращает `std::pair<iterator, bool>`: итератор и признак успеха.
- Сложность: $\mathcal{O}(\log n)$.

④ **`erase(const key_type& k)`**
- Удаляет элемент с ключом `k`, возвращает `0` (если ничего не удалено) или `1` (если удалён элемент).
- Сложность: $\mathcal{O}(\log n)$.

⑤ **`find(const key_type& k)`**

- Возвращает итератор на элемент с ключом `k` или `end()`, если элемент не найден.
- Сложность: $\mathcal{O}(\log n)$.

⑥ **Итераторы**

- `begin()`, `end()`, `rbegin()`, `rend()`, `cbegin()`, `cend()`.
- Тип итераторов — двунаправленный (`bidirectional`).

2.3. Дополнительно

- `std::map` предоставляет методы `lower_bound(k)`, `upper_bound(k)`, `equal_range(k)` для работы с диапазонами ключей.
- Как и `std::set`, `std::map` реализуется поверх сбалансированного двоичного дерева поиска (красно-чёрного).

---

3. `std::multiset`

3.1. Назначение и свойства

- `std::multiset` — это **множество**, в котором **разрешены дубликаты**.
- Использует такой же порядок сортировки, как `std::set` (по умолчанию `std::less<T>` или пользовательский компаратор).
- Основное отличие от `std::set`: повторяющиеся элементы **не запрещены**.

3.2. Основные операции

В целом совпадают с `std::set`, но:

- `insert(val)` может вставить элемент, даже если такое значение уже существует.
- `count(key)` может вернуть число, превышающее `1`, если элементы-дубликаты присутствуют.
- `erase(key)` удаляет **все** копии или только первую? На самом деле перегруженных версий метода несколько:
  - `erase(iterator pos)` — удаляет элемент по итератору.
  - `erase(const key_type& key)` — удаляет **все** элементы, равные `key`. Возвращает их количество.

3.3. Итераторы

- Итераторы — двунаправленные, аналогично `std::set`.
- `begin()`, `end()`, `find()`, `count()`, `lower_bound()`, `upper_bound()`, `equal_range()` — всё аналогично, но учитывает наличие нескольких копий.

---

4. `std::multimap`

4.1. Назначение и свойства

- `std::multimap` — **ассоциативный** контейнер, который допускает несколько пар с одинаковым ключом.
- Это аналог `std::map`, но с возможностью повторных ключей.
- Пары хранятся как `<key, mapped_value>` и упорядочены по ключу (с возможностью пользовательского компаратора).

4.2. Основные операции

- Похожи на `std::map`, но при вызове `insert({k, v})` вставляется новая пара, даже если ключ `k` уже есть.
- `count(key)`, `find(key)`, `lower_bound(key)`, `upper_bound(key)`, `equal_range(key)` могут возвращать результаты, связанные с несколькими элементами, имеющими одинаковый ключ.

4.3. Итераторы

- Двунаправленные итераторы, аналогично `std::map`.

---

5. Итераторы

Все перечисленные контейнеры имеют **двунаправленные (bidirectional)** итераторы:

- Разрешены операции `++it` (переход к следующему элементу) и `--it` (переход к предыдущему).
- Запрещены операции случайного доступа (`it + n`, `it[n]` и т.д.), поэтому нельзя обращаться к элементам по индексу.
- Существуют также `const_iterator`, `reverse_iterator`, `const_reverse_iterator`.

5.1. Примеры базовых методов

- `begin()`, `end()` / `cbegin()`, `cend()` (константные итераторы, C++11+)
- `rbegin()`, `rend()` / `crbegin()`, `crend()` (обратные итераторы, C++11+)

---

6. Компараторы

Что такое компаратор?

Компаратор определяет способ упорядочивания элементов в контейнере (например, ключей в `std::map`).

Пользовательский компаратор

Компаратор можно задать с помощью:

① **Функции:**

```
bool compare(const Key& a, const Key& b) {
   return a > b; // Упорядочивание по убыванию
}
```

① **Функционального класса (функтора):**

```
struct Greater {
    bool operator()(const std::string& lhs, const std::string& rhs) const {
        return lhs > rhs;
    }
};
```

① **Лямбда-функции:**

```
auto greater = [](const std::string& lhs, const std::string& rhs) {
    return lhs > rhs;
```

```
};
```

Пример использования компаратора в std::map

```
#include <iostream>
#include <map>
#include <string>

struct Greater {
    bool operator()(const std::string& lhs, const std::string& rhs) const {
        return lhs > rhs;
    }
};

int main() {
    std::map<std::string, int, Greater> ages = {
        {"Alice", 30},
        {"Bob", 25},
        {"Charlie", 35}
    };

    for (const auto& [key, value] : ages) {
        std::cout << "Key: " << key << ", Value: " << value << std::endl;
    }

    return 0;
}
```

7. Сравнительная таблица

Контейнер	Уникальность ключей	Допустимость дубликатов	Хранимые данные	Доступ по ключу	Итераторы
`std::set`	Уникальные	Нет	Только ключи (`T`)	`find(key)` , `count(key)`	Двунаправленные
`std::map`	Уникальные	Нет	Пары `<Key, T>`	`operator[]` , `at(key)`	Двунаправленные
`std::multiset`	Не уникальные	Да	Только ключи (`T`)	Нет `operator[]`	Двунаправленные
`std::multimap`	Не уникальные	Да	Пары `<Key, T>`	Нет `operator[]`	Двунаправленные

8. Выводы

① `std::set` , `std::multiset` : хранят только ключи. В первом ключи уникальны, во втором допускаются дубликаты.

② `std::map` , `std::multimap` : хранят пары `key-value` . `std::map` требует уникальных ключей, `std::multimap` разрешает повторяющиеся.

③ Все четыре контейнера обеспечивают операции (поиск, вставка, удаление) за $\mathcal{O}(\log n)$ благодаря балансировке на базе красно-чёрных деревьев.

④ Итераторы — двунаправленные. Нет прямого индексационного доступа, так как это ассоциативные структуры.

⑤ Выбор контейнера зависит от того, нужно ли хранить уникальные элементы/ключи и какая форма удобнее (ключи отдельно или в паре).