

# Программирование на C++. Часть 2

## Лекция 10

ПМИ Семестр 2

Демяненко Я.М.

2026

# Ассоциативные контейнеры

- `map<K, V>` — по сути, бинарное дерево поиска, то есть скорость поиска  $\log(n)$
- `set<K>` — множество, так же является бинарным деревом поиска
- `multimap<K, V>` — `map` с повторяющимися ключами
- `multiset<K>` — `set` с повторяющимися значениями
- `unordered_map<K, V>` — хеш-таблица  $O(1)$
- `unordered_set<K>` — хеш-таблица
- `unordered_multiset<K>` — хеш-таблица
- `unordered_multimap <K, V>` — хеш-таблица

У каждой реализации есть свои достоинства и недостатки

Важно, чтобы **базовые операции** для ассоциативных контейнеров:

- вставка insert,
- удаление erase,
- поиск find

выполнялись как **в среднем**, так и **в худшем** случае за время  **$O(\log n)$** .

# Сбалансированные деревья поиска vs хэш-таблицы

Для **сбалансированных** деревьев поиска (в том числе для красно-чёрных деревьев) это условие **выполнено**.

В реализациях, основанных на **хэш-таблицах**, **среднее время** оценивается как  $O(1)$ , что **лучше**, чем в реализациях, основанных на **деревьях поиска**.

Но при этом **не гарантируется высокая скорость** выполнения отдельной операции: **время** операции **вставки в худшем** случае оценивается как  $O(n)$ . Она выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хэш-таблицы.

Хэш-таблицы плохи также тем, что на их основе **нельзя реализовать быстро работающие** дополнительные операции **min**, **max** и алгоритм **обхода всех** хранимых пар в порядке возрастания или **убывания ключей**.

# Ассоциативные контейнеры

**Ассоциативные** контейнеры **обладают многими свойствами**, присущими **последовательным** контейнерам, поскольку они **поддерживают обход элементов данных в виде линейной последовательности**.

Они предоставляют **двунаправленные итераторы**, обход с использованием которых даёт отсортированный порядок элементов в случае с упорядоченными контейнерами.

# Сортировка через мультимножество

В некоторых случаях (например, когда элементы данных представляют собой **большие структуры**) **сортировка** последовательности элементов может быть выполнена более эффективно путем **вставки их в мультимножество и обхода мультимножества**, чем обобщённым алгоритмом сортировки или соответствующей функцией-членом списка.

**Пример.** Даны три множества, содержащих сведения о студентах, неуспевающих по предметам: языки программирования, математический анализ, дифференциальные уравнения. Сведения содержат фамилию, имя, отчество и номер зачётной книжки, используемый в качестве уникального ключа. Найти студентов, неуспевающих по всем трём дисциплинам для формирования приказа на отчисление.

```
class Student {
private:
    string name;
    string recordBook; // номер зачётной книжки (уникальное значение)
public:
    Student(string nm="noname", string rB="*****"): name(nm),recordBook(rB) {}

    friend bool operator<(const Student & p1,const Student & p2){
        return p1.recordBook < p2.recordBook;
    }
    string Name() { return name; }
};
```

# Операции сравнения

Множество **set** не должно содержать повторяющихся значений.

Тем не менее, реализовывать операцию сравнения ключей на равенство не требуется, поскольку проверка на совпадение реализуется через две операции `operator<`.

Определение эквивалентности `x == y` сводится к выражению

$$!(x < y) \&\&! (y < x) \text{ или } !((x < y) \mid \mid (y < x))$$

# Компараторы

Бывают случаи, когда необходимы **сравнения по разным ключам**, а операция `operator<` реализует только одно сравнение.

В этих случаях есть возможность **определить класс для сравнения** значений пользовательского типа.

Такие классы называются **компараторами**.

Например, если сведения о студенте содержат баллы рейтинга `rating`, можно определить компаратор для упорядочения по баллам:

```
class StudentRating{
public:
    bool operator() (const Student & p1, const Student & p2){
        return p1.getrating() < p2.getrating();
    }
}
```

Поскольку у разных студентов могут быть одинаковые баллы, для упорядочения по рейтингу используем **multiset** с двумя параметрами: тип элементов множества и компаратор.

```
multiset<Student, StudentRating > s1;
```

```
#include <iostream>
#include <set>
#include <algorithm>
#include <string>
#include <iterator>
```

```
int main(){
    set<Student> MA, DE, PL;
    // здесь заполняются множества задолжников по: MA, DE, PL, например, с использованием
    операции insert:
    MA.insert(Student("First Student", "MM11111"));
    ...
}
```

Функция-член **insert** классов **set** и **multiset** получает единственный аргумент и **вставляет копию** этого аргумента.

В случае **multiset** элемент всегда вставляется и **результатом вставки** является **позиция** итератора **iterator**, указывающая во множестве на элемент с заданным значением ключа.

Для класса **set** дополнительно указывается, **был ли уже** этот элемент во множестве **или** он был **добавлен** в текущей операции. Поэтому в классе **set** **тип возвращаемого значения** имеет вид **pair<iterator, bool>**.

Использовать такой результат можно следующим образом:

```
set <int> s;
pair<it, bool> a = s.insert(1);           // можно auto a = s.insert(1);
if (a.second)                            // проверяем, вставлен или уже был такой элемент
    cout <<"добавлено " << *(a.first);
else
    cout <<"уже есть во множестве " <<*(a.first);
```

```

set<Student> expelledProj;
insert_iterator<set<Student>> it_ex(expelledProj, expelledProj.begin());

set_intersection(MA.begin(),MA.end(),DE.begin(),DE.end(),it_ex);

for(Student t :expelledProj) {
    cout << t.Name() << " ";
}
cout << endl;

set<Student> expelled;
set_intersection(expelledProj.begin(), expelledProj.end(), PL.begin(), PL.end(), inserter(expelled,
expelled.begin()));

for (Student t : expelled) {
    cout << t.Name() << " ";
}

return 0;
}

```

# Алгоритмы для теоретико-множественных операций

В библиотеке имеются алгоритмы для выполнения теоретико-множественных операций:

```
includes,  
set_union,  
set_intersection,  
set_difference,  
set_symmetric_difference.
```

Эти операции работают с любыми **отсортированными** контейнерами, включая ассоциативные.

В случае неупорядоченных контейнеров алгоритмы для теоретико-множественных операций завершаются ошибкой времени выполнения.

## Пример. Использовать map для подсчёта частоты вхождения слов в текст

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <string>

using namespace std;

int main() {
    ifstream ifs("source.txt");
    if (!ifs.is_open()) {
        cout<<"Не могу открыть файл словаря "<<"source.txt"<<endl;
        exit(1);
    }
}
```

```
typedef istream_iterator<string> string_input;

vector<string> text;
map<string, int> dictionary;

copy(string_input(ifs), string_input(), back_inserter(text));
for (string s: text) {
    ++dictionary[s];
}
for (auto s : dictionary) {
    cout << s.first<<" - "<<s.second;
}
return 0;
}
```

# Операция обращения по индексу

Операция обращения по индексу позволяет работать с `map` как с массивом, используя в качестве индекса ключ.

Если элемента с заданным ключом в контейнере `map` нет, то он добавляется.

При этом ассоциированное по этому ключу значение заполняется значением по умолчанию. Для типа `int` это 0.

Причём это не зависит от того, обращение по индексу входит в `lvalue` или `rvalue` выражения.

В примере это позволяет не делать проверку на существование элемента в контейнере.

В любом случае используется только

**`++dictionary[s];`**

Поскольку при обращении по индексу возможно неявное добавление элемента в `map` с указанием только ключа, то **для типа ассоциированного значения необходим конструктор по умолчанию.**

# Частотный словарь

Просмотр содержимого частотного словаря организуется с помощью цикла `foreach` по словарю.

Автоматически определяемый тип для параметра `s` представляет пару **`pair<string, int>`**.

Доступ к данным в паре осуществляется через поля **`first`** и **`second`**.

```
for (auto s : dictionary) {  
    cout << s.first<<" - "<<s.second;  
}
```

**Пример.** Разработать структуру данных для хранения информации о лекторах, дисциплинах, которые они читают, и форме отчётности по этой дисциплине.

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

typedef map <string, map <string, string>> plan;

int main() {
    plan A;
```

```
map<string,string> p ;
p["ALG"] = "exam";
p.insert(pair<string, string>("LOG", "exam"));
p.insert(pair<string, string>("ML", "credit"));
A.insert(pair<string, map<string, string>>("Hoar", p));
```

```
A["Winner"]["PL"] = "exam";
A["Winner"]["FL"] = "credit";
A["Turing"]["TPL"] = "exam";
A["Turing"]["TML"] = "credit";
A["Virt"] = p;
```

```
for (auto t : A) {
    cout << t.first<<endl;
    for (auto s : t.second) {
        cout << s.first << " " << s.second << endl;
    }
}
return 0;
}
```

Для чего нужны итераторы ?

# Итераторы

Итераторы — **посредники** между контейнерами и обобщенными алгоритмами

Они позволяют создавать обобщённые алгоритмы без учёта того, как именно хранятся последовательности данных, а контейнеры — без необходимости написания большого количества исходного текста работающих с ними алгоритмов

Однако по причинам эффективности **невозможно** обеспечить возможность работы **каждого** обобщённого **алгоритма с каждым контейнером**

Чтобы определить **применимость алгоритма к контейнеру**, необходимо знать **тип итератора**, удовлетворяющего минимальным требованиям алгоритма, и тип итератора, предоставляемый контейнером

# Основные типы итераторов

- итераторы ввода
- итераторы вывода
- однонаправленные (последовательные) итераторы
- двунаправленные итераторы
- итераторы произвольного доступа

# Допустимые для итераторов операции: ++

Тип итератора определяет набор допустимых для него операций.

Для всех типов итераторов определены операции ++ и \*.

Операция ++ имеет одинаковую семантику.

## Допустимые для итераторов операции: \*

Операция \* для итераторов **ввода** возвращает значение элементов коллекции, **не допуская их изменение**.

Для итератора **вывода** операция \* предназначена для изменения элементов коллекции, **не допуская их прочтение**.

Для остальных типов итераторов операция \* позволяет выполнять оба действия над элементами коллекции.

# Допустимые для итераторов операции:

Двунаправленные и произвольного доступа дополнительно имеют операцию  $--$ .

Допустимые для итераторов операции: +, -

Итераторы произвольного доступа, кроме того, имеют операцию смещения на произвольное количество элементов.

# Правила для итераторов

Каждый итератор, имеющий больший набор операций может использоваться вместо более ограниченного в операциях итератора.

Алгоритмы библиотеки STL поддерживают принцип наименьших требований.