**Конспект: Контейнер map**

**map** (словарь, карта) **—** коллекция, сохраняющая пары вида <ключ, значение>. Ключ — некоторая характеризующая значение величина, для которой применима операция сравнения. Пары хранятся в отсортированном виде, что позволяет осуществлять быстрый поиск по ключу, но за это, естественно, придется заплатить: придется так реализовывать вставку, чтобы условие отсортированности не нарушилось.

**Важно.** Ключ должен быть уникальным в пределах словаря. Значения могут повторяться.

Достаточно уникальный контейнер, у которого есть масса применений. Вы знаете что у вас есть уникальный объект, но не уникальное значение, которое ему соответствует. Например: посчитать количество каждой буквы в тексте.

Для работы со словарём нужно **подключить соответствующую библиотеку**:

#include<map>

**Для создания словаря** нужно указать два типа данных в угловых скобках: один для ключа, второй - для значения:

map<int, string> m; // пустой словарь

map<char, int> m1 { // словарь из трёх пар ключ-значение

{‘a’, 1}, {‘b’, 1}, {‘c’, 2}

};

**Добавлять новые элементы** в словарь можно с помощью ключа и метода insert:

m[1] = "a";

m.insert(make\_pair(2, "b"));

По ключу можно менять значение и в принципе **получать доступ к элементам**:

m[2] = "c";

cout << m[2];

**Удалить элементы** из словаря можно с помощью метода erase(n):

m.erase[1]; // удаляем пару с ключом 1

m1.erase[‘c’]; // удаляем пару с ключом с

**Размер словаря** можно определить с помощью метода size():

cout << m.size();

**Метод empty()** возвращает 1, если словарь пуст:

cout << m.empty();

Узнать, содержится ли в словаре пара с определённым ключом можно с помощью методов **count(key)** и **contains(key)**. Первый возвращает количество элементов с определённым ключом, второй - 1 или 0 в зависимости от того, содержится элемент с данным ключом или нет в словаре. Метод contains(key) может не работать, если у вас стандарт языка стоит более ранний, чем С++20.

cout << m.count(1) << m1.count(‘e’);

**Перебор элементов** осуществляется в цикле:

for (auto elem : m)

cout << "Key " << elem.first << ", value " << elem.second << endl;

Начиная со стандарта С++17 можно использовать для перебора пару ключ-значение:

for (const auto& [key, value] : m)

std::cout << key << "\t" << value << std::endl;

**Важно**. Данные в словарях хранятся в отсортированном по ключу виде. Существует также контейнер unordered\_map - неупорядоченный словарь.

**Важно**. В обычных словарях ключи строго уникальны. Если нужно сделать несколько одинаковых ключей, то нужно использовать контейнер multimap - такой словарь, в котором отсутствует условие уникальности ключа, т.е. если вы произведете поиск по ключу, то получите не единственное значение, а набор элементов с одинаковым значением ключа.

#include<iostream>

#include<string>

#include<map> //библиотека для словарей

using namespace std;

int main()

{

map<int, string> m; //словарь, который хранит ключ: число (и) значение: строка

m[392918] = "Maria";

m[392918] = "Ivan"; //можно менять значение по ключу

m[392917] = "Ivan"; //значения могут повторяться, если ключи разные

//m[125522] = "Anna";

m.insert(make\_pair(125522, "Anna")); //еще один способ добавить пару ключ-значение

m[666666] = "Mayvka";

m[1] = "Power";

//словари хранятся осторированными по ключам

for (auto elem : m) {

cout << "Key " << elem.first << ", value " << elem.second << endl;

}

auto iter = m.find(392917); //поиск итератора по ключу

m.erase(iter); //удаление

return 0;

}

Порой бывает что map вообще не следует использовать. Например если существует объект какого то класса и он может сам хранить поле, которое вы могли записать в качестве значения. А контейнер таких объектов можно заменить на наш следующий контейнер

**Пара слов о других контейнерах.**

* **set — это отсортированная коллекция одних только ключей**, т.е. значений, для которых применима операция сравнения, при этом уникальных — каждый ключ может встретиться во множестве (от англ. set — множество) только один раз;

Идеальный вариант для того чтобы использовать только уникальные объекты в проекте

1. Создание множества

#include<iostream>

#include<set> //библиотека для множеств

using namespace std;

int main()

{

set<int> s; //множество, которое хранит целые числа

s.insert(3);

s.insert(1);

s.insert(3);

s.insert(2);

s.insert(3);

s.insert(3);

s.insert(2);

s.insert(4);

s.erase(4); //удалить значение 3

for (auto elem : s) { //range based for

cout << elem << endl;

}

cout <<"first elem >= 2: "<< \*s.lower\_bound(2) << endl;

cout <<"first elem > 2: " << \*s.upper\_bound(2) << endl;

return 0;

}

* **multiset** — **коллекция** с тем же отличием от **set**, что и multimap от map, т.е. **с отсутствием условия уникальности ключа**
* **list — коллекция, хранящая элементы в виде двунаправленного связанного списка.** Не самый частый гость многих проектов, сильно недооценен программистами, которые просто не могут найти списку применение. Самое частое обвинение: Долгий поиск элемента и отсутствие индексов. Но почему-то списки мало кто хвалит за максимальную скорость удаления и добавления элемента в любое место.

Все элементы в списке хранятся в виде указателей друг на друга. первый указывает на второго, второй на первого и третьего, третий на второго и четвертого и так далее. При удалении третьего мы меняем только 2 указателя и это происходит настолько быстро в отличия от вектора, который пересобирает целый, возможно огромный блок в памяти когда ты удалил например только второй элемент.

Используйте list в С++ когда вам нужно часто удалять или добавлять элементы, и не нужен поиск, а тем более индексы

1. Создание двусвязного списка (list)

#include<iostream>

#include<list> //библиотека для двусвязных списков

using namespace std;

int main()

{

list<int> l; //список, который хранит целые числа

l.push\_back(1); //первый элемент

l.push\_front(2); //второй элемент вставляем в начало

l.insert(++l.begin(), 3); //третий элемент вставляем в первый индекс

for (auto elem : l) { //range based for

cout << elem << endl;

}

l.erase(++l.begin()); //удаляем конкретный элемент через итератор

l.pop\_back(); //удаляем последний

l.pop\_front(); //удаляем первый

cout << boolalpha << l.empty() << endl; //пустой ли список?

return 0;

}

**Подробнее об использовании контейнеров можете почитать** [**здесь**](https://codelessons.ru/category/cplusplus/kontejnery) **и** [**здесь**](https://metanit.com/cpp/tutorial/7.1.php)