## Ассоциативный массив С++

Массив – контейнер данных, в котором элементы доступны по целочисленным индексам.

Ассоциативный массив – контейнер данных, в котором элементы доступны по индексам (ключам) имеющим произвольный тип

Множество – вырожденный случай ассоциативного массива, имеющий только ключи (без элементов)

**Телефонный справочник**

#include <iostream>

#include <string>

#include “phonebook.h”

using namespace std;

int

main (){

PhoneBook pBook;

pBook.addRecord (“Vasya”, “121212”);

pBook.addRecord (“Petya”, “212121”);

cout << pBook.getRecord (“Vasya”);

return 0;

}

**Использование класса map**

#include <iostream>

#include <string>

#include <map>

using namespace std;

int

main (){

map <string, string> pBook;

pBook

[“Vasya”] = “121212”;

pBook

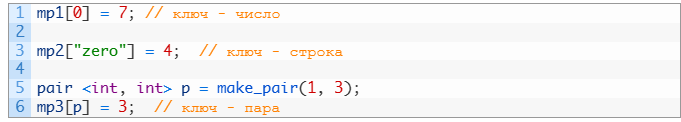
[“Petya”] = “212121”;

cout << pBook [Vasya”];

return 0;

}

В С++ ассоциативные массивы представлены в стандартной библиотеке классом std::map



**Cоздание map**

Сперва понадобится подключить соответствующую библиотеку:

|  |
| --- |
| #include <map> |

Чтобы создать map нужно воспользоваться данной конструкцией:

map < <L>, <R> > <имя>;  
<L> — этот тип данных будет относиться к значению ключа.

<R> — этот тип данных соответственно относится к значению.

map <string, int> mp; // пример

В нашем случае:

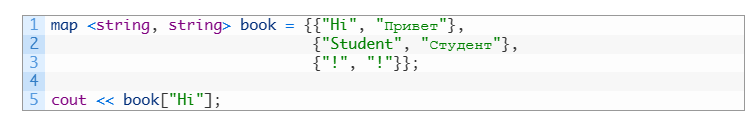
**Ключ** — строка.

**Значение** — число.

При создании map все его элементы будут иметь значение нуля.

Также имеется возможность добавить значения при инициализации (С++ 11 и выше):

**C++**



**Итераторы для map**

Использование итераторов одна из главных тем, если вам понадобится оперировать с этим контейнером. Создание итератора, как обычно происходит так:

map <тип данных> :: iterator <имя>;

* <тип данных> — <string, int> например.

С помощью его можно использовать две операции (it — итератор):

1. Чтобы обратится к ключу нужно сделать так: it->first.
2. Чтобы обратится к значению ячейки нужно сделать так: it->second.

Нельзя обращением к ключу (...->first) изменять его значение, а вот изменять таким образом значение ячейки (...->second) легко.

Нельзя использовать никакие арифметические операции над итератором:

|  |  |
| --- | --- |
| 1  2  3  4  5 | it \*= 5;    it += 3;    it /= 10; |

Все это будет считаться еще одной ошибкой для компилятора.

Для увеличения или уменьшения можно использовать инкремент или декремент.

|  |  |
| --- | --- |
| 1  2 | it++;  it--; |

Нельзя делать то же самое используя операцию присваивания (it += 1) или вычитания (it -= 1).

Чтобы не писать циклы для увеличения итератора на большие значения, можно воспользоваться функцией advance():

|  |  |
| --- | --- |
| 1  2 | advance(it, 7);  advance(it, -5); |

Она сдвигает указанный итератор вниз или вверх на указанное количество ячеек. В нашем случае он сначала увеличит на 7, а потом уменьшит на 5, в итоге получится сдвиг на две вверх.

**Методы** **map**

Ниже мы разберем функции, которые можно использовать для работы с map.

1. ***insert***

Это функция вставки нового элемента.



* num\_1 — ключ.
* num\_2 — значение.

Мы можем сделать то же самое вот так:



1. ***count***

Возвращает количество элементов с данным ключом. В нашем случае будет возвращать — 1 или 0.

Эта функция больше подходит для multimap, у которого таких значений может быть много.



Нужно помнить, что для строк нужно добавлять кавычки — count("Good").

1. ***find***

У этой функции основная цель узнать, есть ли определенный ключ в контейнере.

* Если он есть, то передать итератор на его местоположение.
* Если его нет, то передать итератор на конец контейнера.

Например, давайте разберем данный код:



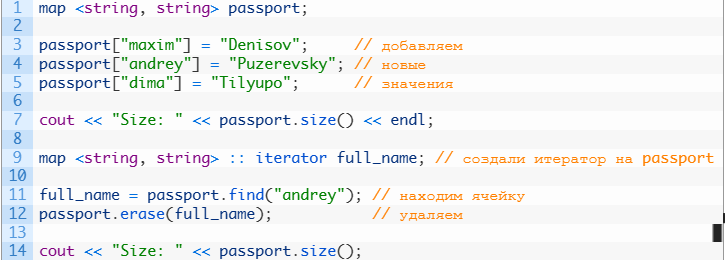
Давайте рассмотрим поподробнее:

* В строке **8**: мы создали map под названием — book.
* В строке **9**: задали единственный ключ "book" со значением "книга".
* В строке **11**: создали два итератора it и it\_2.
* В строке **13 — 14**: получаем итератор на ключ «book» и выводим его значение.
* В строке **16**: получаем итератор на ключ «books», которого нет в нашем контейнере.
* В строке **18 — 20**: проверяем не указывает ли it\_2 на конец контейнера и предупреждаем пользователя.

1. ***erase***

Иногда приходится удалять элементы. Для этого у нас есть функция — erase().

Давайте посмотрим как она работает на примере:



В итоге мы уменьшим количество наших элементов на один.

Также здесь мы воспользовались функцией size(), которая возвращает количество элементов.

passport в нашем случае хранит имя как ключ, а фамилию как значение.

Кроме того класс **map** имеет следующие функции:

* **at()** – возвращает значение по известному ключу;
* **begin()** – возвращает итератор на первый элемент контейнера;
* **clear()** – удаляет из массива все элементы;
* **empty()** – определяет, пуст ли ассоциативный массив (не содержит ни одного элемента);
* **end()** – возвращает итератор, установленный по последнему элементу контейнера;
* **max\_size()** – возвращает размер, фактически выделенный для ассоциативного контейнера с учетом его дальнейшего роста;
* **size()** – возвращает текущее количество элементов (размер) в ассоциативном контейнере;
* **swap()** – обменивает местами содержимое двух контейнеров;
* **operator=()** – присваивает один контейнер другому;
* **operator[]()** – возвращает значение (value) по заданному ключу.

**Источники:**

[Контейнеры: map, set, multiset | brestprog](https://brestprog.by/topics/containers/)

[Map в C++: что это такое и как с ним работать (codelessons.ru)](https://codelessons.ru/cplusplus/map-v-c-chto-eto-i-kak-s-etim-rabotat.html?ysclid=lemoi2h82h568173905)

[Ассоциативные массивы в C++ - GeeksforGeeks (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.4dedeafc-63fc870a-984384ef-74722d776562/https/www.geeksforgeeks.org/associative-arrays-in-cpp/)

## Коллекция Dictionary<K, V>

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Класс словаря Dictionary<K, V> типизируется двумя типами: параметр K представляет тип ключей, а параметр V предоставляет тип значений.

**Создания и инициализация словаря**

Класс Dictionary предоставляет ряд конструкторов для создания словаря. Например, мы можем создать пустой словарь:

Dictionary<int, string> people = new Dictionary<int, string>();

Здесь словарь people в качестве ключей принимает значения типа int, а в качестве значений - строки.

При определении словаря его сразу же можно инициализировать значениями:

var people = new Dictionary<int, string>()

{

{ 5, "Tom"},

{ 3, "Sam"},

{ 11, "Bob"}

};

При инициализации применяется инициализитор - в фигурных скобках после вызова конструктора объекту передаются начальные данные. В случае со словаем мы можем передать в инициализаторе набор элементов, где каждый элемент заключается в фигурные скобки, например:

{ 5, "Tom"}

Каждый элемент представляет два значения: первое значение представляет ключ, а второе значение - собственно значение элемента. Поскольку при объявлении словаря people для ключей указан тип int, а для значений - тип string, то в элементе словаря сначала указывается число int, а затем строка. То есть в случае выше элемент имеет ключ 5, а значение - "Tom". Затем по ключу элемента мы сможем получить его значение.

Также мы можем применять другой способ инициализации:

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob"

};

При таком способе инициализации в квадратных скобках указывается ключ и ему присваивается значение элемента. Но в целом этот способ инициализации будет равноценен предыдущему.

**KeyValuePair**

Стоит отметить, что каждый элемент в словаре представляет структуру ***KeyValuePair<TKey, TValue>,*** где параметр ***TKey*** представляет тип ключа, а параметр ***TValue*** - тип значений элементов. Эта структура предоставляет свойства Key и Value, с помощью которых можно получить соответственно ключ и значение элемента в словаре. И одна из версий конструктора Dictionary позволяет инициализировать словарь коллекцией объектов KeyValuePair:

var mike = new KeyValuePair<int, string>(56, "Mike");

var employees = new List<KeyValuePair<int, string>>() { mike};

var people = new Dictionary<int, string>(employees);

Конструктор типа KeyValuePair принимает два параметра - ключ элемента и его значения. То есть в данном случае создается один такой элемент - mike с ключом 56 и значением "Mike". И этот элемент добавляется в список employees, которым затем инициализируется словарь.

Можно совместить оба способа инициализации:

var mike = new KeyValuePair<int, string>(56, "Mike");

var employees = new List<KeyValuePair<int, string>>() { mike };

var people = new Dictionary<int, string>(employees)

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob",

};

В данном случае в словаре people будет четыре элемента.

**Перебор словаря**

Для перебора словаря можно применять цикл foreach:

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob"

};

foreach(var person in people)

{

Console.WriteLine($"key: {person.Key} value: {person.Value}");

}

При переборе каждый элемент будет помещаться в переменную, которая представляет тип KeyValuePair, соответственно с помощью свойств Key и Value мы сможем получить ключ и значение элемента. Консольный вывод программы:

key: 5 value: Tom

key: 6 value: Sam

key: 7 value: Bob

**Получение элементов**

Для обращения к элементам из словаря применяется их ключ, который передается в квадратных скобках:

словарь[ключ]

Таким образом мы можем получить и изменить элементы словаря

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob",

};

// получаем элемент по ключу 6

string sam = people[6]; // Sam

Console.WriteLine(sam); // Sam

// переустанавливаем значение по ключу 6

people[6] = "Mike";

Console.WriteLine(people[6]); // Mike

// добавляем новый элемент по ключу 22

people[22] = "Eugene";

Console.WriteLine(people[22]); // Eugene

Более того, таким образом мы можем также добавить новый элемент в словарь. При установке значения по ключу, если элемент с таким ключом уже есть в словаре, то значение переустанавливается. Если же элемента с подобным ключом нет в словаре, то элемент добавляется.:

**Методы и свойства Dictionary**

Среди методов класса Dictionary можно выделить следующие:

void Add(K key, V value): добавляет новый элемент в словарь

* **void Clear():** очищает словарь
* **bool ContainsKey(K key):** проверяет наличие элемента с определенным ключом и возвращает true при его наличии в словаре
* **bool ContainsValue(V value):** проверяет наличие элемента с определенным значением и возвращает true при его наличии в словаре
* **bool Remove(K key):** удаляет по ключу элемент из словаря.

Другая версия этого метода позволяет получить удленный элемент в выходной параметр: bool Remove(K key, out V value)

* **bool TryGetValue(K key, out V value):** получает из словаря элемент по ключу key. При успешном получении передает значение элемента в выходной параметр value и возвращает true
* **bool TryAdd(K key, V value):** добавляет в словарь элемент с ключом key и значением value. При успешном добавлении возвращает true

Из свойств следует отметить свойство Count, которое возвращает количество элементов в словаре.

Применение методов:

// условная телефонная книга

var phoneBook = new Dictionary<string, string>();

// добавляем элемент: ключ - номер телефона, значение - имя абонента

phoneBook.Add("+123456", "Tom");

// альтернативное добавление

// phoneBook["+123456"] = "Tom";

// Проверка наличия

var phoneExists1 = phoneBook.ContainsKey("+123456"); // true

Console.WriteLine($"+123456: {phoneExists1}");

var phoneExists2 = phoneBook.ContainsKey("+567456"); // false

Console.WriteLine($"+567456: {phoneExists2}");

var abonentExists1 = phoneBook.ContainsValue("Tom"); // true

Console.WriteLine($"Tom: {abonentExists1}");

var abonentExists2 = phoneBook.ContainsValue("Bob"); // false

Console.WriteLine($"Bob: {abonentExists2}");

// удаление элемента

phoneBook.Remove("+123456");

// проверяем количество элементов после удаления

Console.WriteLine($"Count: {phoneBook.Count}"); // Count: 0