

Неделя 2

Функции и контейнеры

2.1. Функции

2.1.1. Объявление функции. Возвращаемое значение.

Прежде код программы записывался внутри функции `main`. В данном уроке будет рассмотрено, как определять функции в C++. Для начала, рассмотрим преимущества разбиения кода на функции:

- Программу, код которой разбит на функции, проще понять.
- Правильно подобранное название функции помогает понять ее назначение без необходимости читать ее код.
- Выделение кода в функцию позволяет его повторное использование, что ускоряет написание программ.
- Функции — это единственный способ реализовать рекурсивные алгоритмы.

Теперь можно приступить к написанию первой функции. Объявление функции содержит:

- Тип возвращаемого функцией значения.
- Имя функции.
- Параметры функции. Перечисляются через запятую в круглых скобках после имени функции. Для каждого параметра нужно указать не только его имя, но и тип.
- Тело функции. Расположено в фигурных скобках. Может содержать любые команды языка C++. Для возврата значения из функции используется оператор `return`, который также завершает выполнение функции.

Например, так выглядит функция, которая возвращает сумму двух своих аргументов:

```
int Sum(int x, int y) {  
    return x + y;  
}
```

Чтобы воспользоваться функцией, достаточно вызвать ее, передав требуемые параметры:

```
int x, y;  
cin >> x >> y;  
cout << Sum(x, y);
```

Данный код считывает два значения из консоли и выведет их сумму.

Важной особенностью оператора `return` является то, что он завершает выполнение функции. Рассмотрим функцию, проверяющую, входит ли слово в некоторый набор слов:

```
bool Contains(vector<string> words, string w) {  
    for (auto s : words) {  
        if (s == w) {  
            return true;  
        }  
    }  
    return false;  
}
```

Функция принимает набор строк и строку `w`, для которой надо проверить, входит ли она в заданный набор. Возвращаемое значение — логическое значение (входит или не входит), имеет тип `bool`.

Результат работы функции для нескольких случаев:

```
cout << Contains({"air", "water", "fire"}, "fire"); // 1  
cout << Contains({"air", "water", "fire"}, "milk"); // 0  
cout << Contains({"air", "water", "fire"}, "water"); // 1
```

Функция работает так, как ожидалось. Стоит отметить, что в C++ значения логического типа выводятся как 0 и 1. Чтобы лучше понять, как выполнялась функция, запустим отладчик.

Далее представлен код программы с указанием номеров строк и результат пошагового исполнения в виде таблицы. Первый столбец — номер шага, второй — строка, которая выполняется на данном шаге, а третий — значение переменной `s`, определенной внутри функции.

```

1  #include <iostream>
2  #include <vector>
3  #include <string>
4
5  using namespace std;
6
7  bool Contains(vector<string> words, string w)
   ↪ {
8      for (auto s : words) {
9          if (s == w) {
10             return true;
11         }
12     }
13     return false;
14 }
15
16 int main() {
17     cout << Contains({"air", "water", "fire"},
   ↪ "water") << endl; // 1
18     return 0;
19 }

```

№	LN	string s
0	16	-
1	17	-
2	7	-
3	8	air
4	9	air
5	8	water
6	9	water
7	10	water
8	17	-
9	18	-

Видно, что программа в цикле успевает перебрать только первые два значения из вектора, после чего возвращает `true` и выполнение функции прекращается. Этот пример демонстрирует то, что оператор `return` завершает выполнение функции.

Ключевое слово `void`

Рассмотрим функцию, которая выводит на экран некоторый набор слов, который был передан в качестве параметра.

```

??? PrintWords(vector<string> words) {
    for (auto w : words) {
        cout << w << " ";
    }
} /* */

```

Остается вопрос: что следует написать в качестве типа возвращаемого значения. Функция по своей сути ничего не возвращает и не понятно, какой тип она должна возвращать.

В случаях, когда функция не возвращает никакого значения, в качестве возвращаемого типа используется ключевое слово `void` (*англ.* пустой). Таким образом, код функции будет следующим:

```
void PrintWords(vector<string> words) {  
    for (auto w : words) {  
        cout << w << " ";  
    }  
}
```

После того, как функция была определена, ее можно вызвать:

```
PrintWords({"air", "water", "fire"})
```

2.1.2. Передача параметров по значению

Рассмотрим следующую функцию, которая была написана, чтобы устанавливать значение 42 передаваемому ей аргументу:

```
void ChangeInt(int x) {  
    x = 42;  
}
```

В функции `main` эта функция вызывается с переменной `a` в качестве параметра, значение которой до этого было равно 5.

```
int a = 5;  
ChangeInt(a);  
cout << a;
```

После вызова функции `ChangeInt` значение переменной `a` выводится на экран. Вопрос: что будет выведено на экран, 5 или 42?

Верный способ проверить — запустить программу и посмотреть. Запустив ее, можно убедиться, что программа выводит «5», то есть значение переменной `a` внутри `main` не поменялось в результате вызова функции `ChangeInt`.

Этот пример призван продемонстрировать то, что параметры функции передаются по значению. Другими словами, в функцию передаются копии значений, переданные ей во время вызова.

Посмотрим на то, как это происходит с помощью пошагового выполнения.

```
1  #include <iostream>
2  using namespace std;
3
4  void ChangeInt(int x) {
5      x = 42;
6  }
7
8  int main() {
9      int a = 5;
10     ChangeInt(a);
11     cout << a;
12     return 0;
13 }
```

№	LN	int a	int x
0	9	-	-
1	10	5	-
2	4	5	5
3	5	5	42
4	11	5	-

Видно, что меняется значение переменной внутри функции `ChangeInt`, а значение переменной в `main` остается тем же.

2.1.3. Передача параметров по ссылке

Поскольку параметры функции передаются по значению, изменение локальных формальных параметров функции не приводит к изменению фактических параметров. Объекты, которые были переданы функции на месте ее вызова, останутся неизменными. Но что делать в случае, если функция по смыслу должна поменять объекты, которые в нее передали.

Допустим, нужно написать функцию, которая обменивает значения двух переменных. Проверим, подходит ли такая функция для этого:

```
void Swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Определим две переменные, `a` и `b`:

```
int a = 1;
int b = 2;
```

От правильно работающей функции ожидается, что значения переменных поменяются местами, а именно переменная `a` будет равна двум, а `b` — одному. Применим функцию `Swap`.

```
Swap(a, b);
```

Выведем на экран значения переменных:

```
cout << "a == " << a << '\n'; // 1
cout << "b == " << b << '\n'; // 2
```

Мы видим, что значения переменных не изменились. Действительно, поскольку параметры функции при вызове были скопированы, изменение переменных `x` и `y` никак не привело к изменению переменных внутри функции `main`.

Чтобы реализовать функцию `Swap` правильно, параметры `x` и `y` нужно передавать по ссылке. Это соответствует тому, что в качестве типа параметров нужно указывать не `int`, а `int&`. После исправления функция принимает вид:

```
void Swap(int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Запустив программу снова, можно убедиться, что функция отработала так, как и ожидалось.

Таким образом, для модификации передаваемых в качестве параметров объектов, их нужно передавать не по значению, а по ссылке. Ссылка — это особый тип языка C++, является синонимом переданного в качестве параметра объекта. Ссылка оформляется с помощью знака `&` после типа передаваемой переменной.

Можно привести еще один пример, в котором оказывается полезным передача параметров функции по ссылке. Уже говорилось, что в библиотеке `algorithm` существует функция сортировки. Например, отсортировать вектор из целых чисел можно так:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};
sort(begin(nums), end(nums));
```

Чтобы проверить, что все работает, также выведем элементы вектора на экран:

```
for (auto x : nums) {
    cout << x << " ";
}
```

Запускаем программу. Программа выводит: «0 1 2 2 3 6», то есть вектор отсортировался.

Однако, у данного способа есть недостаток: при вызове `sort` дважды указывается имя вектора, что увеличивает вероятность ошибки из-за невнимательности при написании кода. В результате опечатки программа может не скомпилироваться или, что гораздо хуже, работать неправильно. Поэтому хотелось бы написать такую функцию сортировки, при вызове которой имя вектора нужно указывать лишь раз.

Без использования ссылок такая функция выглядела бы примерно так:

```
vector<int> Sort(vector<int> v) {  
    sort(begin(v), end(v));  
    return v;  
}
```

Она принимает в качестве параметра вектор из целых чисел и возвращает также вектор целых чисел, а внутри выполняет вызов функции `sort`. Запустим программу:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};  
nums = Sort(nums);  
for (auto x : nums) {  
    cout << x << " ";  
}
```

Убеждаемся, что программа дает тот же результат. Но мы не избавились от дублирования: мало того, что в месте вызова мы также указываем имя вектора дважды, так и в определении функции `Sort` тип вектора указывается также два раза.

Перепишем функцию, используя передачу параметра по ссылке:

```
void Sort(vector<int>& v) {  
    sort(begin(v), end(v));  
}
```

Такая функция уже ничего не возвращает, а изменяет переданный ей в качестве параметра объект. Поэтому при ее вызове указывать имя вектора нужно один раз:

```
vector<int> nums = {3, 6, 1, 2, 0, 2};  
Sort(nums);  
for (auto x : nums) {  
    cout << x << " ";  
}
```

Именно это и хотелось получить.

2.1.4. Передача параметров по константной ссылке

Раньше было показано, как в C++ можно создавать свои типы данных. А именно была определена структура Person:

```
struct Person {  
    string name;  
    string surname;  
    int age;  
};
```

Допустим, что была проведена перепись Москвы и вектор из Person, который содержит в себе данные про всех жителей Москвы, можно получить с помощью функции GetMoscowPopulation:

```
vector<Person> GetMoscowPopulation();
```

Здесь специально не приводится тело этой функции, которое может быть устроено очень сложно, отправлять запросы к базам данных и так далее. Вызвать эту функцию можно так:

```
vector<Person> moscow_population = GetMoscowPopulation();
```

Требуется написать функцию, которая выводит на экран количество людей, живущих в Москве. Эта функция ничего не возвращает, принимает в качестве параметра вектор людей и выводит красивое сообщение:

```
void PrintPopulationSize(vector<Person> p) {  
    cout << "There are " << p.size() <<  
        " people in Moscow" << endl;  
}
```

Воспользуемся этой функцией:

```
vector<Person> moscow_population = GetMoscowPopulation();  
PrintPopulationSize(moscow_population);
```

Программа вывела, что в Москве 12500000 людей: «There are 12500000 people in Moscow».

Замерим время выполнение функции GetMoscowPopulation и функции PrintPopulationSize. Подключим специальную библиотеку для работы с промежутками времени, которая называется chrono:

```
#include <chrono>  
#include <iostream>  
#include <vector>  
#include <string>  
  
using namespace std;  
using namespace std::chrono;
```


После этого до и после места вызова каждой из интересующих функций получим текущее значение времени, а затем выведем на экран разницу:

```
auto start = steady_clock::now();
vector<Person> moscow_population = GetMoscowPopulation();
auto finish = steady_clock::now();
cout << "GetMoscowPopulation "
    << duration_cast<milliseconds>(finish - start).count()
    << " ms" << endl;

start = steady_clock::now();
PrintPopulationSize(moscow_population);
finish = steady_clock::now();
cout << "PrintPopulationSize "
    << duration_cast<milliseconds>(finish - start).count()
    << " ms" << endl;
```

В результате получаем:

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 1034 ms
```

Получается, что функция, которая возвращает вектор из 12 миллионов строк, работает быстрее функции, которая всего-то печатает размер этого вектора. Функция PrintPopulationSize ничего больше не делает, но работает дольше.

Но мы уже говорили, что при передаче параметров в функции происходит полное глубокое копирование передаваемых переменных, в данном случае — вектора из 12 500 000 элементов. Фактически, чтобы вывести на экран размер вектора, мы тратим целую секунду на его полное копирование. С этим нужно как-то бороться.

Избежать копирования можно с помощью передачи параметров по ссылке:

```
void PrintPopulationSize(vector<Person>& p) {
    cout << "There are " << p.size() <<
        " people in Moscow" << endl;
}
```

```
GetMoscowPopulation 609 ms
There are 12500000 people in Moscow
PrintPopulationSize 0 ms
```

Теперь все работает хорошо, но у данного способа есть несколько недостатков:

- Передача параметра по ссылке — способ изменить переданный объект. Но в данном случае функция не меняет объект, а просто печатает его размер. Объявление этой функции

```
void PrintPopulationSize(vector<Person>& p)
```

может сбивать с толку. Может создаться впечатление, что функция как-то меняет свой аргумент.

- В случае, если промежуточная переменная не создается:

```
PrintPopulationSize(GetMoscowPopulation());)
```

программа даже не скомпилируется. Дело в том, что в C++ результат вызова функции не может быть передан по ссылке в другую функцию (почему это так будет сказано позже в курсе).

Получается, что при передаче по значению, мы вынуждены мириться с глубоким копированием всего вектора при каждом вызове функции печати размера, а при передаче по ссылке — мириться с вышеназванными двумя проблемами. Существует ли идеальное решение без всех этих недостатков?

Выход заключается в использовании передачи параметров по так называемой константной ссылке. Это делается с помощью ключевого слова **const**, которое добавляется слева от типа параметра. Символ **&** остается на месте и указывает, что происходит передача по ссылке.

Определение функции принимает вид:

```
void PrintPopulationSize(const vector<Person>& p) {  
    cout << "There are " << p.size() <<  
        " people in Moscow" << endl;  
}
```

В результате `PrintPopulationSize` выполняется за 0 мс, а также работает передача результата вызова функции в качестве параметра другой функции по константной ссылке:

```
PrintPopulationSize(GetMoscowPopulation());)
```

Также мы не вводим в заблуждение пользователей нашей функции и явно указываем, что параметр не будет изменен, так как он передается по константной ссылке.

Такой подход также защищает от случайного изменения фактических параметров функций. Допустим, по ошибке в функцию печати количества

людей в Москве попал код, добавляющий туда одного жителя Санкт-Петербурга.

```
void PrintPopulationSize(const vector<Person>& p) {
    cout << "There are " << p.size() <<
        " people in Moscow" << endl;
    p.push_back({"Vladimir", "Petrov", 40});
}
```

В случае передачи по ссылке такая ошибка могла бы остаться незамеченной, но при передаче по константной ссылке такая программа даже не скомпилируется:

```
main.cpp: In function 'void PrintPopulationSize(const std::vector<
    Person>&)':
main.cpp:20:41: error: passing 'const std::vector<Person>' as 'this
    ' argument discards qualifiers [-fpermissive]
    p.push_back({"Vladimir", "Petrov", 40});
                                ^
```

Компилятор в таком случае выдает ошибку, так как нельзя изменять принятые по константной ссылке фактические параметры.

2.2. Модификатор `const` как защита от случайных изменений

На самом деле `const` — специальный модификатор типа переменной, запрещающий изменение данных, содержащихся в ней.

Например, рассмотрим следующий код:

```
int x = 5;
x = 6;
x += 4;
cout << x;
```

В этом коде переменная `x` изменяется в двух местах. Как несложно убедиться, в результате будет выведено «10». Добавим ключевое слово `const`, не меняя ничего более:

```
const int x = 5;
x = 6;
x += 4;
cout << x;
```

При попытке скомпилировать этот код, компилятор выдает следующие сообщения об ошибках:

```
main.cpp: In function 'int main()':
main.cpp:9:7: error: assignment of read-only variable 'x'
    x = 6;
    ^
main.cpp:10:8: error: assignment of read-only variable 'x'
    x += 4;
    ^
```

Обе строчки, в которых переменная подвергается изменению, приводят к ошибкам. Закомментируем их:

```
const int x = 5;
//x = 6;
//x += 4;
cout << x;
```

Теперь программа компилируется как надо и выводит «5». Чтение переменной является немодифицирующей операцией и не вызывает ошибок при компиляции.

Рассмотрим пример со строковой переменной `s`:

```
string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s << endl;
```

Здесь представлены операции: получение длины строки, добавление текста в конец строки, инициализация другой строки значением `s+'!'`, вывод значения строки в консоль. Запускаем программу:

```
5
hello, world
hello, world!
```

Теперь добавляем модификатор `const`.

```
const string s = "hello";
cout << s.size() << endl;
s += ", world";
string t = s + "!";
cout << s;
```

При компиляции только в одном месте выводится ошибка:

```
basic_string.h:1131:7: note:   in call to 'std::__cxx11::
basic_string<_CharT, _Traits, _Alloc>& std::__cxx11::
basic_string<_CharT, _Traits, _Alloc>::operator+=(const _CharT*)
[with _CharT = char; _Traits = std::char_traits<char>; _Alloc =
std::allocator<char>]'
    operator+=(const _CharT* __s)
    ~~~~~~
```

Вывод длины строки, использование строки при инициализации другой строки и вывод строки в консоль — немодифицирующие операции и ошибок не вызывают. А вот добавление в конец строки еще как-либо текста — уже нет. Закомментируем соответствующую строку:

```
const string s = "hello";
cout << s.size() << endl;
//s += ", world";
string t = s + "!";
cout << s;
```

```
5
hello
hello!
```

Более сложный пример: рассмотрим вектор строк и попытаемся изменить первую букву первого слова этого вектора с прописной на заглавную:

```
vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

Программа успешно компилируется и выводит «Hello» как и ожидалось. Установим модификатор `const`.

```
const vector<string> w = {"hello"};
w[0][0] = 'H';
cout << w[0];
```

В итоге компиляция завершается ошибкой:

```
main.cpp:9:13: error: assignment of read-only location '(& w.std::
vector<std::__cxx11::basic_string<char> >::operator [] (0))->std::
__cxx11::basic_string<char>::operator [] (0) '
w[0][0] = 'H';
```

Здесь важно отметить следующее: мы не меняем вектор непосредственно (не добавляем элементы, не меняем его размер), а модифицируем только его элемент. Но в C++ модификатор `const` распространяется и на элементы контейнеров, что и демонстрируется в данном примере.

Зачем вообще в C++ нужен модификатор `const`?

Главное предназначение модификатора `const` — помочь программисту не допускать ошибок, связанных с ненамеренными модификациями переменных. Мы можем пометить ключевым словом `const` те переменные, которые не хотим изменять, и компилятор выдаст ошибку в том месте, где происходит ее изменение. Это позволяет экономить время при написании кода, так как избавляет от мучительных часов отладки.

2.3. Контейнеры

2.3.1. Контейнер vector

Тип `vector` представляет собой набор элементов одного типа. Тип элементов вектора указывается в угловых скобках. Классический сценарий использования вектора — сохранение последовательности элементов.

Создание вектора требуемой длины. Ввод и вывод с консоли

Напишем программу, которая считывает из консоли последовательность строк, например, имен лекторов. Сначала на вход подается число элементов последовательности:

```
int n;  
cin >> n;
```

Поскольку известно количество элементов последовательности, его можно указать в конструкторе вектора (то есть в круглых скобках после названия переменной):

```
vector<string> v(n);
```

После этого можно с помощью цикла `for` перебрать все элементы вектора по ссылке:

```
for (string& s : v) {  
    cin >> s;  
}
```

Каждый очередной элемент `s` — ссылка на очередной элемент вектора. С помощью этой ссылки считывается очередная строка.

Теперь остается вывести вектор на экран, чтобы проверить, что все было считано правильно. Для этого удобно написать специальную функцию, которая выводит все значения вектора. Вызываем функцию следующим образом:

```
PrintVector(v);
```

А само определение функции `PrintVector` располагаем над функцией `main`:

```
void PrintVector(const vector<string>& v) {  
    for (string s : v) {  
        cout << s << endl;  
    }  
}
```

Запустим программу и проверим, что она работает:

```
> 2
> Anton
> Ilia
Anton
Ilia
```

Отлично: мы успешно считали элементы вектора и успешно их вывели.

Добавление элементов в вектор. Методы `push_back` и `size`

Можно реализовать эту программу несколько иначе с помощью цикла `while`. Также считаем число элементов вектора `n`, но создадим пустой вектор `v`.

```
int n;
cin >> n;
vector<string> v;
```

Создадим переменную `i`, в которой будет храниться индекс считываемой на данной итерации строки.

```
int i = 0;
```

В цикле `while` считываем строку из консоли в локальную вспомогательную переменную `s`, которая добавляется к вектору с помощью метода `push_back`:

```
while (i < n) {
    string s;
    cin >> s;
    v.push_back(s);
    cout << "Current size = " << v.size() << endl;
    ++i;
}
```

В конце каждой итерации значение `i` увеличивается на 1. Чтобы продемонстрировать, что размер вектора меняется, на каждой итерации его текущий размер выводится на экран.

После завершения цикла чтения, как и в предыдущем примере, выводим значения вектора на экран с помощью функции `PrintVector`:

```
PrintVector(v);
```

```
> 2
> first
Current size = 1
> second
Current size = 2
first
second
```

Как и ожидалось, после ввода первой строки текущий размер стал равным 1, а после ввода второй — равным 2.

Вернемся к прошлой программе, в которой размер вектора задавался через конструктор в самом начале, и добавим туда также вывод размера на каждой итерации цикла:

```
int n;
cin >> n;
vector<string> v(n);
for (string& s: v) {
    cin >> s;
    cout << "Current size = " << v.size() << endl;
}
PrintVector(v);
```

Запустим программу и убедимся, что в таком случае размер вектора постоянен:

```
> 2
> first
Current size = 2
> second
Current size = 2
first
second
```

Так происходит, потому что в самом начале программы вектор создается сразу нужного размера.

Задание элементов вектора при его создании

Бывают случаи, когда содержимое вектора заранее известно. В этом случае указать заранее известные значения при создании вектора можно с помощью фигурных скобок. Например, числовой вектор, содержащий количество дней в каждом месяце (для краткости: в первых 5 месяцах), можно создать так:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};
```

Такой вектор можно распечатать:

```
PrintVector(days_in_months);
```


Правда, сперва следует подправить функцию PrintVector так, чтобы она принимала числовой вектор, а не вектор строк:

```
void PrintVector(const vector<int>& v) {  
    for (auto s : v) {  
        cout << s << endl;  
    }  
}
```

Запустим программу, убеждаемся, что она работает как надо.

Иногда бывает необходимым изменить значения вектора после его создания. Например, в високосных годах количество дней в феврале — 29, и чтобы это учесть, слегка допишем нашу программу:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};  
if (true) { // if year is leap  
    days_in_months[1]++;  
}  
PrintVector(days_in_months);
```

Здесь для простоты проверка на високосность опущена. Замечу, что в C++ элементы вектора нумеруются с нуля, поэтому количество дней в феврале хранится в первом элементе вектора.

Из этого примера можно сделать вывод, что вектор также можно использовать для хранения элементов в привязке к их индексам.

Создание вектора, заполненного значением по умолчанию

Допустим, нужно создать вектор, который для каждого дня в феврале хранит, является ли данный день праздничным. В этом случае следует использовать вектор булевых значений. Поскольку большинство дней праздничными не являются, хотелось бы, чтобы при создании вектора все его значения по умолчанию были false.

Значение по умолчанию можно указать, передав его в качестве второго аргумента конструктора:

```
vector<bool> is_holiday(28, false);
```

В качестве первого аргумента конструктора указывается длина вектора, как и в первом примере. После этого заполним элементы вектора. Например, известно, что 23 февраля — праздничный день:

```
is_holiday[22] = true;
```

Вывести вектор в консоль можно с помощью функции PrintVector:

```
PrintVector(is_holiday);
```

Функцию PrintVector все же предстоит сперва доработать, чтобы она принимала вектор булевых значений.

```
void PrintVector(const vector<bool>& v) {  
    for (auto s : v) {  
        cout << s << endl;  
    }  
}
```

Заметим, что изменилось только определение типа при задании параметра функции, а ее тело осталось неизменным. В будущем это позволит обобщить эту функцию для вывода векторов разных типов. Но пока мы не обсудили этот вопрос, приходится довольствоваться только функциями, каждая из которых работает с векторами определенного типа.

Изменение длины вектора

Для удобства сперва доработаем функцию вывода, чтобы кроме значений выводились также и индексы элементов.

```
void PrintVector(const vector<bool>& v) {  
    int i = 0;  
    for (auto s : v) {  
        cout << i << ": " << s << endl;  
        ++i;  
    }  
}
```

Иногда бывает необходимым изменить длину вектора. Например, если необходимо (по тем или иным причинам) созданный в предыдущей программе вектор использовать для хранения праздничных мартовских дней, его нужно сперва расширить и заполнить значением по умолчанию.

Попытаемся сделать это с помощью функции `resize`, которая может выполнить то, что надо. Попробуем это сделать:

```
is_holiday.resize(31);  
PrintVector(is_holiday);
```

Метод `resize` сделал не то, что мы хотели, потому что старые значения остались и 23 марта оказалось праздничным. Если мы хотим переиспользовать этот вектор и сделать его нужной длины, нам понадобится метод `assign`:

```
is_holiday.assign(31, false);
```

В качестве первого аргумента передается желаемый размер вектора, а в качестве второго — какими элементами проинициализировать его элементы. Теперь можно указать, что 8 марта — праздничный день:

```
is_holiday[7] = true;
```

Запустив код, убеждаемся, что «упоминание о 23 марта» пропало, как и хотелось.

```
PrintVector(is_holiday);
```

Очистить вектор можно с помощью метода `clear`:

```
is_holiday.clear();
```

2.3.2. Контейнер `map`

Создание словаря. Добавление элементов

Допустим, требуется хранить важные события в привязке к годам, в которые они произошли. Для решения этой задачи лучше всего подходит такой контейнер как словарь. Словарь состоит из пар ключ-значение, причем ключи не могут повторяться. Для работы со словарями нужно подключить соответствующий заголовочный файл:

```
#include <map>
```

Создадим словарь с ключами типа `int` и строковыми значениями:

```
map<int, string> events;  
events[1950] = "Bjarne Stroustrup's birth";  
events[1941] = "Dennis Ritchie's birth";  
events[1970] = "UNIX epoch start";
```

Напишем функцию, которая позволяет вывести словарь на экран:

```
void PrintMap(const map<int, string>& m) {  
    cout << "Size = " << m.size() << endl;  
    for (auto item: m) {  
        cout << item.first << ": " << item.second << endl;  
    }  
}
```

Обратиться к ключу очередного элемента `item` при итерировании можно как `item.first`, а к значению — как к `item.second`. Также добавим в функцию вывода словаря вывод его размера (используя метод `size`).

Итерирование по элементам словаря

Выведем получившийся словарь на экран с помощью написанной функции:

```
PrintMap(events);
```

На экран будут выведены три элемента:

```
Size = 3
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
1970: UNIX epoch start
```

Словарь не просто вывелся на экран в формате ключ-значение. В выводе ключи оказались отсортированными в порядке возрастания целых чисел.

Этот пример демонстрирует одно из важных свойств словаря: элементы в нем хранятся отсортированными по ключам, а также выводятся отсортированными в цикле `for`.

Обращение по ключу к элементам словаря

Кроме того, можно обращаться к конкретным значениям из словаря по ключу. Например, можно узнать событие, которое произошло в 1950 году:

```
cout << events[1950] << endl;
```

```
Bjarne Stroustrup's birth
```

Отдельно отметим, что такой синтаксис очень напоминает синтаксис для получения значения элемента вектора по индексу. В некотором смысле, словарь позволил расширить функционал вектора: теперь в качестве ключей можно указывать сколь угодно большие целые числа.

Удаление по ключу элементов словаря

Элементы словаря можно не только добавлять в него, но и удалять. Для удаления элемента словаря по ключу используется метод `erase`:

```
events.erase(1970);
PrintMap(events);
```

```
Size = 2
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
```

Построение «обратного» словаря

Ключи словаря могут иметь тип `string`. Продемонстрируем это, обратив построенный нами словарь. Словарь, который получится в результате, позволит получать по названию события год, когда это событие произошло.

Для построения такого словаря, напомним функцию `BuildReversedMap`:

```
map<string, int> BuildReversedMap(
    const map<int, string>& m) {
    map<string, int> result;
    for (auto item: m) {
        result[item.second] = item.first;
    }
    return result;
}
```

Реализация этой функции достаточно проста. Сперва нужно приготовить итоговый словарь, типы ключей и значений в котором переставлены по сравнению с исходным словарем. Затем в цикле `for` нужно пробежаться по всем элементам исходного словаря и записать в итоговый, используя в качестве ключа бывшее значение, а в качестве значения — ключ. После цикла нужно вернуть получившийся словарь с помощью `return`.

Для вывода на экран получившегося словаря необходимо написать функцию `PrintReversedMap`, поскольку мы пока не научились писать функцию, выводящую на печать словарь любого типа:

```
void PrintReversedMap(const map<string, int>& m) {
    cout << "Size = " << m.size() << endl;
    for (auto item: m) {
        cout << item.first << ": " << item.second << endl;
    }
}
```

Еще раз отметим, что тело функции уже довольно общее и в нем нигде не содержатся типы ключей и значений.

Теперь можно запустить следующий код:

```
map<string, int> event_for_year = BuildReversedMap(events);
PrintReversedMap(event_for_year);
```

```
Size = 2
Bjarne Stroustrup's birth: 1950
Dennis Ritchie's birth: 1941
```

Также по названиям событий можно получить год, в котором они произошли:

```
cout << event_for_year["Bjarne Stroustrup's birth"];
```

```
1950
```

Создание словаря по заранее известным данным

Создание словаря по заранее известному набору пар ключ-значение можно произвести следующим образом с помощью фигурных скобок:

```
map<string, int> m = {{"one", 1}, {"two", 2}, {"three", 3}};
```

Выведем словарь на экран и убедимся, что он создан правильно:

```
PrintMap(m);
```

```
one: 1
three: 3
two: 2
```

Все ключи здесь отсортировались лексикографически, то есть в алфавитном порядке.

Следует также отметить, что функцию печати словаря можно улучшить, итерируясь по нему по константной ссылке:

```
void PrintMap(const map<string, int>& m) {
    for (const auto& item: m) {
        cout << item.first << ": " << item.second << endl;
    }
}
```

В таком случае получается избежать лишнего копирования элементов словаря.

Еще раз отметим, как удалять значения из словаря, например для ключа «three»:

```
map<string, int> m = {{"one", 1}, {"two", 2}, {"three", 3}};
m.erase("three");
PrintMap(m);
```

```
one: 1
two: 2
```

Подсчет количества различных элементов последовательности

Словари могут быть полезными, если необходимо подсчитать, сколько раз встречаются элементы в некоторой последовательности.

Допустим, дана последовательность слов:

```
vector<string> words = {"one", "two", "one"};
```


Строки могут повторяться. Необходимо подсчитать, сколько раз встретилась каждое слово из этой последовательности. Для этого создадим словарь:

```
map<string, int> counters;
```

После этого пробежимся по всем элементам последовательности. Случай, когда слово еще не встречалось, нужно будет рассматривать отдельно, например так:

```
for (const string& word : words) {
    if (counters.count(word) == 0) {
        counters[word] = 1;
    } else {
        ++counters[word];
    }
}
```

Проверка на то, содержится ли элемент в словаре, может быть произведена с помощью метода count, как показано в коде. Такой код, безусловно, работает, но оказывается, что он избыточен. Достаточно написать так:

```
 for (const string& word : words) {
    PrintMap(counters);
    ++counters[word];
}
```

- PrintMap(counters);

Дело в том, что как только происходит обращение к конкретному элементу словаря с помощью квадратных скобок, компилятор уже создает пару для этого ключа со значением по умолчанию (для целого числа значение по умолчанию — 0).

Здесь мы сразу добавили вывод всего словаря для того, чтобы продемонстрировать как меняется размер словаря (в функцию PrintMap также добавлен вывод размера словаря):

```
Size = 0
Size = 1
one: 1
```

```
Size = 2
one: 1
two: 1
Size = 2
one: 2
two: 1
```

Продemonстрируем, что от простого обращения к элементу словаря происходит добавление к нему пары с этим ключом и значением по умолчанию:

```
map<string, int> counters;
counters["a"];
PrintMap(counters);
```

```
Size = 1
a: 0
```

Группировка слов по первой букве

Приведем еще один пример, показывающий, как можно использовать свойство изменения размера словаря при обращении к несуществующему ключу. Предположим, что необходимо сгруппировать слова из некоторой последовательности по первой букве. Решение данной задачи может выглядеть следующим образом:

```
vector<string> words = {"one", "two", "three"};
map<char, vector<string>> grouped_words;
for (const string& word : words) {
    grouped_words[word[0]].push_back(word);
}
```

В цикле for сначала идет обращение к несуществующему ключу (первой букве каждого слова). При этом ключ добавляется в словарь вместе с пустым вектором в качестве значения. Далее, с помощью метода `push_back` текущее слово присваивается в качестве значения текущего ключа. Выведем словарь на экран и убедимся, что слова были сгруппированы по первой букве:

```
for (const auto& item: grouped_words) {
    cout << item.first << endl;
    for (const string& word : item.second) {
        cout << word << " ";
    }
    cout << endl;
}
```



```
o
one
t
two three
```

Стандарт C++17

Недавно комитет по стандартизации языка C++ утвердил новый стандарт C++17. Говоря простым языком, были утверждены новые возможности языка. Но, к сожалению, изменения в стандарте только спустя некоторое время отражаются в свежих версиях компиляторов. Также свежие версии компиляторов не всегда просто использовать, так как они еще не появились в дистрибутивах для разработки. Тем не менее, имеет смысл рассказывать о свежих возможностях языка, даже если пока они не поддерживаются компиляторами и их еще нельзя использовать.

Чтобы попробовать новые возможности компиляторов, существуют различные ресурсы, например gsc.goldbolt.org. Он представляет собой окно ввода кода на C++ и панель для выбора версии компилятора. На данной панели можно выбрать еще не вышедшую версию компилятора gsc 7. Чтобы сказать компилятору, что код будет соответствовать новому стандарту, нужно указать флаг компиляции `|--std=c++17|`.

Среди новых возможностей — новый синтаксис для итерирования по словарю. Например, так бы выглядел код итерирования с использованием старого стандарта:

```
#include <map>

using namespace std;

int main() {
    map<string, int> m = {{"one", 1}, {"two", 2}};
    for (const auto& item : m) {
        item.first, item.second;
    }

    return 0;
}
```

В данном коде имеются следующие проблемы:

- Переменная `item` имеет «странный» тип с полями `first` и `second`.
- Нужно либо помнить, что `first` соответствует ключу, а `second` — значению текущего элемента, либо заводить временные переменные.

В новом стандарте появляется возможность писать такой код более понятно:

```
map<string, int> m = {{"one", 1}, {"two", 2}};
for (const auto& [key, value] : m) {
    key, value;
}
```

2.3.3. Контейнер set

Допустим, необходимо сохранить для каждого человека, является ли он известным. В этом случае можно было бы завести словарь, ключами в котором были бы строки, а значениями — логические значения:

```
map<string, bool> is_famous_person;
```

Теперь, чтобы указать, что какие-то люди являются известными, можно написать следующий код:

```
is_famous_person["Stroustrup"] = true;
is_famous_person["Ritchie"] = true;
```

Имеет ли смысл добавлять в этот словарь людей, которые являются неизвестными? Наверное, нет: таких людей слишком много и их нет нужды хранить, когда можно хранить только известных людей. А в этом случае значениями в таком словаре являются только true.

Создание множества. Добавление элементов.

Для решения такой задачи более естественно использовать другой контейнер — множество (set). Для работы с множествами необходимо подключить соответствующий заголовочный файл:

```
#include <set>
```

Теперь можно создать множество известных людей:

```
set<string> famous_persons;
```

Добавить в это множество элементы можно с помощью метода insert:

```
famous_persons.insert("Stroustrup");
famous_persons.insert("Ritchie");
```

Печать элементов множества

Функция PrintSet, позволяющая печатать на экране все элементы множества строк, реализуется следующим образом:

```
void PrintSet(const set<string>& s) {  
    cout << "Size = " << s.size() << endl;  
    for (auto x : s) {  
        cout << x << endl;  
    }  
}
```

В эту функцию сразу добавлен вывод размера множества — он может быть получен с помощью метода size.

Теперь можно вывести на экран элементы множества известных людей:

```
PrintSet(famous_persons);
```

```
Size = 2  
Ritchie  
Stroustrup
```

Элементы множества выводятся в отсортированном порядке, а не в порядке добавления.

Также гарантируется уникальность элементов. То есть повторно никакой элемент не может быть добавлен в множество.

```
set<string> famous_persons;  
famous_persons.insert("Stroustrup");  
famous_persons.insert("Ritchie");  
famous_persons.insert("Stroustrup");  
PrintSet(famous_persons);
```

```
Size = 2  
Ritchie  
Stroustrup
```

Удаление элемента

Удаление из множества производится с помощью метода erase:

```
set<string> famous_persons;  
famous_persons.insert("Stroustrup");  
famous_persons.insert("Ritchie");  
famous_persons.insert("Anton");  
PrintSet(famous_persons);
```

```
Size = 3
Anton
Ritchie
Stroustrup
```

```
famous_persons.erase("Anton");
PrintSet(famous_persons);
```

```
Size = 2
Ritchie
Stroustrup
```

Создание множества с известными значениями

С помощью фигурных скобок можно создать множество, заранее указывая значения содержащихся в нем элементов. Например, множество названий месяцев может быть инициализировано как:

```
set<string> month_names =
    {"January", "March", "February", "March"};
PrintSet(month_names);
```

```
Size = 3
February
January
March
```

Сравнение множеств

Как и другие контейнеры, множества можно сравнивать:

```
set<string> month_names =
    {"January", "March", "February", "March"};
set<string> other_month_names =
    {"March", "January", "February"};

cout << (month_names == other_month_names) << endl;
```

В результате будет выведено «1», то есть эти множества равны.

Проверка принадлежности элемента множеству

Для того, чтобы быстро проверить, принадлежит ли элемент множеству, можно использовать метод count:

```
set<string> month_names =
    {"January", "March", "February", "March"};
cout << month_names.count("January") << endl;
```

Создание множества по вектору

Чтобы создать множество по вектору, не обязательно писать цикл. Реализовать это можно следующим образом:

```
vector<string> v = {"a", "b", "a"};  
set<string> s(begin(v), end(v));  
PrintSet(s);
```

Size = 2

a

b

С помощью аналогичного синтаксиса можно создать и вектор по множеству.