IX

Стандартная библиотека C++. Контейнеры

Типы для хранения набора значений произвольных типов.

Контейнеры

Под контейнером понимается некоторый тип, в смысле языка программирования, объекты которого могут хранить в себе более одного значения. Примерами *встроенных* в C++ контейнеров могут служить статическией массива (массивы неизменной длины):

```
1 // давным-давно, в далёкой далёкой pdf'ке:
2 double my_arr[4] = {1.2, 3.4, 5.6, -7.8};
```

Другие типы контейнеров реализуются на самом языке C++ и входят в стандартную библиотеку. Далее будут расмотренны: **динамический массив**, **статический массив** (как объект), **ассоциативные массивы**, **множества** и некоторые другие. Технически, все эти контейнеры реализованы через *шаблонные классы*. И могут работать как со всеми **фундаментальными** типами, так и с большинством **составных** типов.

Контейнеры. Динамический массив

Динамический массив представлен в C++ шаблонным классом **vector**. Для его использования следует подключить следующий заголовочный файл:

```
1 #include <vector>
```

Общая форма для задания объектов данного класса есть:

```
1 #include <vector>
2
3 vector<Type> var_name( args... );
```

, где **Type** - любой тип данных, **var_name** - имя переменной, **args...** - аргументы, передаваемые в конструктор.

Прежде, чем идти дальше

Все контейнеры стандартной библиотеки C++ помещены в *пространство имён* **std**, но далее практически во всех примерах явное указание **std::** пропущено.

Контейнеры. Динамический массив

Динамический массив: основные конструкторы, общая форма:

```
1 // (1)
2 vector<Type> var1()
3
4 // (2)
5 vector<Type> var2(size_t count)
6
7 // (3)
8 vector<Type> var2(size_t count, Type value)
```

1. **(1)** - конструктор без параметров, просто создаёт массив нулевой длины.
2. **(2)** - создаём массив и выделяем место под **count** элементов. Начальные значения элементам не присваиваются.
3. **(3)** - создаём массив под **count** элементов и **каждому из них** присваиваем значение **value**.

Контейнеры. Динамический массив

Динамический массив: основные конструкторы, примеры:

```
1  // массив целых чисел, нулевой длины
2  vector<int> int_array;
3
4  // массив чисел с плавающей точкой на 10 значений
5  vector<double> real_array(10);
6
7  string base_value = "ABC";
8  // массив строк, содержащий 5 элементов,
9  // каждый из которых равен строке "ABC"
10 vector<string> str_array(5, base_value);
```

Вызова конструктора в примерах делается через круглые скобки из-за того, что для типа **vector** (да и других контейнеров) дополнительно определён конструктор, реализующий **инициализацию списком** значений.

Технически, этот конструктор для типа **vector** принимает параметром список значений (через параметр типа **initializer_list**):

```
1 // Создаётся массив целых чисел, который после
2 // создания состоит из 6 элементов, каждому из
3 // которых присвоено соответствующее значение
4 // из фигурных скобок справа
5 vector<int> int_arr2{1, 5, 6, 7, 8, 10};
```

Напомним, что при наличии конструкторов с параметрами и инициализации списком в некоторых случаях имеется неоднозначность:

```
1 // Массив из двух элементов или массив из
2 // восьми элементов, каждый равен 101 ?
3 vector<int> int_arr3{8, 101};
```

Технически, этот конструктор для типа **vector** принимает параметром список значений (через параметр типа **initializer_list**):

```
1 // Создаётся массив целых чисел, который после
2 // создания состоит из 6 элементов, каждому из
3 // которых присвоено соответствующее значение
4 // из фигурных скобок справа
5 vector<int> int_arr2{1, 5, 6, 7, 8, 10};
```

Напомним, что при наличии конструкторов с параметрами и инициализации списком в некоторых случаях имеется неоднозначность:

```
1 // Массив из двух элементов или массив из
2 // восьми элементов, каждый равен 101 ?
3 vector<int> int_arr3{8, 101};
```

Ответ: **int_arr** будет массивом из двух значений, 8 и 101.

Контейнеры. Динамический массив

Поэтому именно для динамического массива, в частности, и для всех контейнеров в стандартной библиотеке C++, определена простая рекомендация:

Круглые скобки рулят для контейнеров

Для разрешения необнозначности при вызове конструкторов класса **vector** и ему подобных предпочитайте ***круглые скобки***.

Кроме того, напомним, что в следующем примере создания объектов динамических массивов обе строки вызывают один и тот же конструктор:

```
1 vector<int> int_arr2{1, 5, 6, 7, 8, 10};
2
3 vector<int> int_arr5 = {1, 5, 6, 7, 8, 10};
```

Правила инициализации они такие.

Динамический массив::методы информационные

```
vector<Type> my_arr(10);

(1) size_t my_arr.size();
(2) size_t my_arr.max_size();
(3) bool my_arr.empty();
```

- **(1)** - узнать текущий размер массива;
- **(2)** - узнать потенциально максимальное количество элементов , которые может хранить массив;
- **(3)** - метод возращает **true** если массив не содержит ни одного элемента, **false** - в противоположном случае.

Динамический массив::методы, изменяющие объект

```
vector<Type> my_arr(10);

(4) void my_arr.resize(size_t new_size);
    void my_arr.resize(size_t new_size, type val);
(5) void my_arr.reserve(size_t count);
(6) void my_arr.clear();
```

4. **(4)** - поменять размер массива на **new_size**. Если **new_size** меньше текущего размера - лишние элементы удаляются. Если больше - то выделяется память под нужное количество элементов. С помощью **val** - добавляемым элементам можно задать конкретное начальное значение
5. **(5)** - если **count** больше текущего размера массива, под недостающие элементы выделяется память
6. **(6)** - удалить все элементы из массива

Динамический массив::примеры использования

```
1  vector<int> int_arr, int_arr2(14, 5);
2
3  string base_value = "ABC";
4  vector<string> str_arr(5, base_value);
5
6  cout << "\nРазмер str_arr: " << str_arr.size();
7  cout << std::boolalpha;
8  cout << "\nint_arr пуст? " << int_arr.empty();
9
10 str_arr.resize(10, "mmm");
11 cout << "\nРазмер str_arr: " << str_arr.size();
12
13 int_arr2.reserve(20);
14 cout << "\nРазмер int_arr2: " << int_arr2.size();
```

Динамический массив::доступ к элементам

```
(1) Type& my_arr[size_t n];
(2) Type& my_arr.at(size_t n);
```

1. **(1)** - получить ссылку на элемент с индексом **n**
2. **(2)** - получить ссылку на элемент с индексом **n**

```
1 vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3 int_arr[0] = 8;
4 cout << "\nПервый элемент равен: " << int_arr[0];
5
6 int_arr.at(3) = 14;
7 cout << "\nЧетвёртый: " << int_arr.at(3);
```

Динамический массив::доступ к элементам

```
(1) Type& my_arr[size_t n];
(2) Type& my_arr.at(size_t n);
```

Разница между **(1)** и **(2)** способами обращения к элементу массивы заключается в том, что метод **at** проверяет тот факт, что переданный индекс не выходит за границу массива. Если всё-таки выходит, что *выбрасывается* исключение **out_of_range**. При обращении к элементу через оператор «**квадратные скобки**» поведение неопределено. Как правило, произойдёт обращение к блоку памяти вне выделенного динамического массива.

Динамический массив::доступ к элементам

```
(1) Type& my_arr[size_t n];
(2) Type& my_arr.at(size_t n);
```

На примере массива со слайда 13:

```
8  // Поведение неопределено,
9  // вероятно случится ошибка обращения к памяти:
10 cout << "\nНеизвестный элемент: " << int_arr[3001];
11
12 // Обработка исключений демонстрирует разницу
13 try {
14   cout << "\nДругая попытка: " << int_arr.at(3001);
15 }
16 catch (std::out_of_range & ex) {
17   cout << ex.what();
18 }
```

Динамический массив::доступ к элементам

```
(3) Type& my_arr.front();
(4) Type& my_arr.back();
```

- **(3)** - получить ссылку на первый элемент
- **(4)** - получить ссылку на последний элемент

```
1  vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3  cout << "Первый элемент: " << int_arr.front();
4  cout << "Последний элемент: " << int_arr.back();
5
6  int_arr.front() = 25;
7  int_arr.back() += 10;
8
9  cout << "Первый элемент: " << int_arr.front();
10 cout << "Последний элемент: " << int_arr.back();
```

Динамический массив::итераторы

```
(1) iterator my_arr.begin();
    iterator begin(my_arr);
(2) iterator my_arr.end();
    iterator end(my_arr);
```

- **(1)** - получить итератор, указывающий на первый элемент массива. В библиотеке <**vector**> итератор определён и как через метод класса, и как свободная функция;
- **(2)** - получить итератор, указывающий на элемент массива, **следующий за последним**.

Пример сортировки:

```
1 vector<int> int_arr = {10, 8, 3, 5, -4, 5, 3};
2 std::sort(int_arr.begin(), int_arr.end());
3 for (const auto& elem : int_arr) {
4   cout << elem << ", ";
5 }
6 cout << endl;
```

Динамический массив::итераторы

Тип итератора для **vector** определён как

```
1 vector<int> int_arr = {10, 8, 3, 5, -4, 5, 3};
2 vector<int>::iterator it = int_arr.begin();
3
4 cout << "Доступ через итератор к первому элементу: "
5      << *it << endl;
6 cout << "И второму: " << *(it + 1) << endl;
```

Подобным образом определены итераторы (название класса, двоеточие, слово **iterator**) для большинства контейнеров стандартной библиотеки C++.

Динамический массив::итераторы

Пример: применение функции **find** из библиотеки **<algorithm>**

```
1  vector<int> iarr = {10, 8, 3, 5, -4, 5, 3};
2  int to_search;
3
4  cout << "Введите элемент для поиска: ";
5  cin >> to_search;
6
7  vector<int>::iterator found;
8  found = find(begin(iarr), end(iarr), to_search);
9
10 if (found != iarr.end()) {
11     cout << "Найдено значение: " << *found << endl;
12 } else {
13     cout << to_search << " не найден в массиве\n";
14 }
```

Функция **find** как раз возвращает итератор на элемент массива, если его значение совпадает со значениями третьего параметра. Когда совпадение не найдено, возвращается итератор **arr.end()**

Динамический массив::итераторы

Помня об общей концепции итераторов, предыдущий пример упрощается с использованием **auto**

```
1  vector<int> iarr = {10, 8, 3, 5, -4, 5, 3};
2  int to_search;
3
4  cout << "Введите элемент для поиска: ";
5  cin >> to_search;
6  // не надо явно указывать vector_type::iterator
7  auto found = find(begin(iarr), end(iarr), to_search);
8
9  if (found != iarr.end()) {
10     cout << "Найдено значение: " << *found << endl;
11 } else {
12     cout << to_search << " не найден в массиве\n";
13 }
```

Другие функции из <**algorithm**>, возвращающие итераторы: **search**, **search_n**, **find_end**. См. примеры тут: http://www.cplusplus.com/reference/algorithm/

Динамический массив::добавление элементов

```
(1) void my_arr.push_back(const Type& value );
(2) template<typename... Args>
    void my_arr.emplace_back(Args&& ...args);
(3) template<typename... Args>
    iter my_arr.emplace(iter pos, Args&& ...args);
```

- **(1)** - добавить элемент **value** в конец массива (путём копирования);
- **(2)** - создать элемент используя аргумент(ы) для конструктора, переданные через специальный параметр **args**;
- **(3)** - создать элемент, используя **args**, и вставить его на позицию, заданную итератором **pos**. Данный метод возвращает итератор, указывающий на добавленный элемент массива.

Динамический массив::добавление элементов

Использование **push_back**:

```
1  vector<int> iarr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3  iarr.push_back(888);
4  iarr.push_back(777);
5  cout << "Последний элемент: " << iarr.back() << endl;
6
7
8  iarr.push_back(-1);
9  iarr.push_back(-3);
10 iarr.push_back(-5);
11 iarr.push_back(-7);
12 iarr.push_back(-9);
13
14 cout << "А теперь: " << iarr.back() << endl;
```

Динамический массив::добавление элементов

emplace и **emplace_back**: используя выше упомянутый параметр **args** они создают (конструируют) элемент массива конкретного типа данных и добавляют его либо в конец массива, либо на позицию, заданную итератором. Для фундаментальных типов отличий от **push_back** немного:

```
1  vector<int> iarr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2  iarr.emplace_back(-8);
3  iarr.emplace_back(-7);
4  cout << "Последний элемент: " << iarr.back() << endl;
5  // Создаём итератор на 5-й элемент
6  auto it = iarr.begin() + 4;
7  iarr.emplace(it, 200);
8
9  cout << "Массив iarr: ";
10 for (const auto& elem : iarr) {
11   cout << elem << ", ";
12 }
13 cout << endl;
```

Динамический массив::добавление элементов

emplace и **emplace_back**: но разница есть, когда используется динамический массив некоторых объектов, которые имеют нетривиальный конструктор:

```
1  class JustTest
2  {
3  public:
4      // Класс с пользовательским конструктором
5      JustTest(int value, bool invert, string s = "↩
           default") :
6          i_field{value}, s_field{s}
7      {
8        if (invert) {
9          i_field *= -1;
10       }
11     }
12
13     int i_field;
14     string s_field;
15 };
```

Динамический массив::добавление элементов

emplace и **emplace_back**: и для объектов класса выше создадим массив:

```
1 vector<JustTest> test_arr;
2
3 test_arr.emplace_back(10, true, "строка разумная");
4 test_arr.emplace_back(3, false);
5 test_arr.emplace_back(-15, true, "yet another str");
6
7 cout << test_arr[2].i_field << ", [["
8      << test_arr[2].s_field << "]]" << endl;
```

В строках **(3)** - **(5)** методу **emplace_back** передаются ровно те значения, которые необходимы конструктору класса **JustTest**. Шаблонный класс **vector** может хранить объекты любых пользовательских типов, которые не запрещают операцию копирования содержимого своих объектов.

Динамический массив::удаление элементов

```
(1) Type& my_arr.pop_back();
(2) iter my_arr.erase(iter pos);
    iter my_arr.erase(iter start, iter end);
```

- **(5)** - удалить последний элемент
- **(6)** - первая форма: удалить элемент, который стоит на позиции, указываемой итератором **pos**. Вторая форма: удалить элементы в диапазоне **[start; end)**, где **start** - итератор на первый **удаляемый** элемент, **end** - итератор на первый неудаляемый элемент (т.е. удаляются все элементы вплоть до **end - 1**). Обе формы возвращают итератор, указывающий на значение **my_arr.end()** после удаления элемента.

Динамический массив::удаление элементов

```
1  vector<int> iarr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3  iarr.pop_back();
4  cout << "Последний элемент: " << iarr.back() << endl;
5
6  auto third = iarr.begin() + 2;
7  iarr.erase(third); // Удаляем третий элемент
8
9  auto fourth = iarr.begin() + 3,
10      seventh = iarr.begin() + 6;
11 // Удалить элементы с 4-го по 6-ой
12 iarr.erase(fourth, seventh);
13
14 cout << "Массив iarr: ";
15 for (const auto& elem : iarr) {
16   cout << elem << ", ";
17 }
18 cout << endl;
```

Динамический массив::разрастание размерности

Динамический массив: с помощью шаблонного класса **vector** можно задавать многомерные массива. Но при этом при обращении к элементам надо самостоятельно следить, что эти элементы действительно существуют:

```
1  vector<vector<int>> int2D = { {1, 2, 3},
2                                {4, 5, 6},
3                                {7, 8, 9, 10} };
4
5  cout << "Длина первой строки: " << int2D[0].size()
6       << endl;
7  cout << "Длина последней: " << int2D[2].size()
8       << endl;
9
10 cout << "int2D[1][1] = " << int2D[1][1] << endl;
11
12 // Так делать НЕ надо (правда же?):
13 int2D[5][4] = 888;
14 cout << "Что-нибудь: " << int2D[20][102] << endl;
```

Динамический массив::операции сравнения

Доступны операторы сравнения: проверка на равенство, больше, меньше, больше или равно, меньше или равно. Работают они следующим образом: сначала сравниваются размеры двух массивов, затем, если они равны, выполняется поэлементная операция сравнения. Если каждая поэлементная операция вернула **true**, общий результат будет таким же. Если хоть одно поэлементное сравнение вернуло **false**, это же значение и будет результатом сравнения массивов. На примерах:

```
1 vector<int> iarr1 = {1, 2, 3}, iarr2 = {3, 2, 3};
2 vector<int> iarr3 = {1, 2, 3};
3
4 if (iarr1 == iarr2) {
5   cout << "iarr1 и iarr2 равны\n";
6 }
7 if (iarr1 == iarr3) {
8   cout << "iarr1 и iarr3 равны\n";
9 }
```

Статический массив::сущность

Стандартная библиотека C++ предоставляет шаблонный класс **array** для создания и манипуляции массивами **неизменяемой** длины как объектами (а не уже известными встроенными типами). Заголовочный файл для подключения: <**array**>. Общая форма создания такого массива:

```
1 array<Type, size_t count> stat_array;
```

где

- **Type** - тип хранимых элементов;
- **count** - размер статического массива. Второй параметр шаблона должен быть константой времени компилирования программы.

Данный шаблонный класс предоставляет **конструктор без параметров** и конструктор, позволяющий инициализировать элементы массива списком значений переменной длины.

Статический массив::доступ к элементам

Такие же возможности, что и для динамического массива:

```
(1) Type& my_arr[pos];
(2) Type& my_arr.at(pos);
(3) Type& my_arr.front();
(4) Type& my_arr.back();
```

```
1 array<int, 5> stat_arr = {10, 9, 8, 7, 6};
2
3 stat_arr[3] = 5;
4 cout << "Четвёртый элемент: " << stat_arr[3] << endl;
5
6 stat_arr.back() = 900;
7 cout << "Последний: " << stat_arr.back() << endl;
```

Статический массив::знания об объекте

```
(1) size_t my_arr.size();
(2) bool my_arr.empty();
```

- **(1)** - метод для получения длины статического массива (хотя из контекста обычно очевидно);
- **(2)** - проверка, имеет ли статический массив длину **0**.

```
1 // Да, так можно:
2 array<int, 0> arr1;
3 array<int, 1> arr2;
4
5 cout << std::boolalpha;
6 cout << "arr1 пустой? " << arr1.empty() << endl;
7 cout << "arr2 пустой? " << arr2.empty() << endl;
```

Статический массив::сравнение

Для **array** также определены операторы сравнения. Этим он в некоторых ситуациях является значительно удобнее, чем встроенные в язык статические массивы. Но есть важный момент: размер массива тоже определяет тип **array**. Это означает, что C++ не позволит сравнивать два статических массива, которые хранят элементы одного типа, но имеют разную длину.

```
1  array<int, 5> arr1 = {10, 20, 30, 40, 50};
2  array<int, 5> arr2 = {10, 20, 70, 40, 50};
3  array<int, 7> arr3 = {10, 20, 30, 40, 50, 60, 70};
4
5  cout << std::boolalpha;
6  cout << "arr1 равен arr2: " << (arr1 == arr2) << endl;
7  cout << "arr1 меньше arr2: " << (arr1 < arr2) << endl;
8  cout << "arr1 больше arr2: " << (arr1 > arr2) << endl;
9
10 // Ошибка компиляции:
11 // cout << "arr1 == arr3: " << (arr1 == arr3) << endl;
```

Статический массив::передача в функции

Объекты статического массива (как и динамического) могут передаваться в функции как **по значению**, так и **по ссылке**:

```
1 void by_copy(array<int, 10> arr)
2 {
3   // что-то полезное...
4 }
5
6 void by_reference(array<int, 10>& arr)
7 {
8   // что-то полезное...
9 }
```

Но и тут длина должна быть указана для конкретизации типа передаваемого объекта.

Статический массив::и тут итераторы

Шаблонный тип **array** предоставляет полный набор итераторов. Поэтому **for-range**, функции из **<algorithm>** без проблем раотают с объектами рассматриваемого шаблонного класса:

```
1  array<int, 5> arr1 = {10, -20, 30, -40, 50};
2
3  sort(arr1.begin(), arr1.end());
4
5  cout << "Массив по возрастанию: ";
6  for (const auto& elem : arr1) {
7    cout << elem << ", ";
8  }
9  cout << endl;
10
11 auto end_it = arr1.end(),
12      found = find(arr1.begin(), end_it, -20);
13 if (found != end_it) {
14     cout << "Значение " << *found
15          << " присутствует в arr1\n";
16 }
```

Контейнеры::очередь и стек

В стандартной библиотеке C++ содержатся файлы <**queue**> и <**stack**>, представляюие собой шаблонные классы очереди и стека, соответственно. Оба класса являются динамическими, работают для произвольного типа хранимых элементов.
Для обоих контейнеров есть общая особенность: они не предоставляют никаких итераторов для обхода своего содержимого (это противоречит абстрактному определению очереди и стека).
Базовая работа с очередью/стеком показана на следующих двух примерах. Подробнее предоставляемый интерфейс этих классов можно посмотреть тут:
http://www.cplusplus.com/reference/queue/queue/
http://www.cplusplus.com/reference/stack/stack/

Контейнеры::очередь

```
1  #include <queue> // нужная библиотека
2
3  queue<int> my_queue;
4  int num;
5
6  cout << "Вводите целые числа (0 - для прекращения)\n";
7  do {
8    cin >> num;
9    if (num == 0) { break; }
10
11   my_queue.push(num);
12 } while (true);
13
14 cout << "Введённая очередь:\n";
15 while ( !my_queue.empty() ) {
16   cout << my_queue.front() << ' ';
17   my_queue.pop();
18 }
```

Контейнеры::стек

```
1  #include <stack> // нужная библиотека
2
3  stack<int> my_stack;
4  int num;
5
6  cout << "Вводите целые числа (0 - для прекращения)\n";
7  do {
8    cin >> num;
9    if (num == 0) { break; }
10
11   my_stack.push(num);
12 } while (true);
13
14 cout << "Введённый стек:\n";
15 while ( !my_stack.empty() ) {
16   cout << my_stack.top() << ' ';
17   my_stack.pop();
18 }
```

Контейнеры::пара значений

Пара значений представлена в C++ шаблонной структурой **pair**. Хранит в себе два значения любой комбинации двух типов данных. Для его использования следует подключить следующий заголовочный файл:

```
1 #include <utility>
```

Общая форма для задания объектов данного класса есть:

```
1 #include <utility>
2
3 pair<Type1, Type2> var_name( args... );
```

, где **Type1** - тип данных первого значения, **Type2** - тип данных второго значения, **var_name** - имя переменной, **args...** - аргументы, передаваемые в конструктор.

Пара значений::конструкторы

```
(1) pair<Type1, Type2> my_pair()
(2) pair<Type1, Type2> my_pair(Type1 & val1,
                               Type2 & val2)
```

- **(1)** - конструктор без параметров, просто создаёт экземпляр структуры с двумя полями, не присваивая никаких начальных значений созданному объекту
- **(2)** - создаём экземпляр структуры; первое поле получает значение **val1**, второе - **val2**

```
1 pair<int, double> pair1;
2 pair<int, char> pair2(35, 'D');
```

Пара значений::доступ к полям

```
template <typename Type1, typename Type2>
struct pair
{
  Type1 first;
  Type2 second;
};
```

```
1  pair<int, double> pair1;
2  pair<int, char> pair2(35, 'D'), pair3;
3
4  cout << "\nПервое значение pair2: " << pair2.first;
5
6  pair1.second = 15.888;
7  cout << "\nВторое значение pair1: " << pair1.second;
8
9  pair3 = pair2; // Копирование
10 pair3.first = 55;
11 cout << "\nПервое значение pair3: " << pair3.first;
```

Пара значений::функция для создания

Возможно создание объектов с помощью шаблонной функции **make_pair** (также объявлена в **<utility>**)

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(T1 & val1, T2 & val2)
```

```
1  pair<int, double> pair1;
2  pair1 = std::make_pair(555, 0.783);
3
4  cout << "\nЗначения pair1: " << pair1.first
5                               << ' '
6                               << pair1.second;
7
8  // Использование auto для вывода типа:
9  auto pair2 = std::make_pair(808, -1.7123);
10 cout << "\nЗначения pair2: " << pair2.first
11                              << ' '
12                              << pair2.second;
```

Ассоциативный массив

Ассоциативный массив - специальный тип данных, в котором индексом массива может быть объект произвольного типа. Известен также по терминам «**хеш**», «**мап**» и «**словарь**» в различных языках программирования. Суть можно выразить следующим псевдокодом:

```
1 cool_arr["str as index"] = MaterialPoint{1, 2, 3, 2.3};
```

Здесь операция индексации осталась (как в привычных статических или динамических массивах), но индексом служит уже не целое число. Объект в квадратных скобках называется **ключём** ассоциативного массива, а присваеваемый этому ключу объект - его(ключа) **значением**.
В стандартной библиотеке C++ ассоциативный массив реализован через шаблонные типы, которые позволяют задать разные типы для ключа и значения.

Ассоциативный массив

Стоит заметить

Ассоциативные массивы в C++ по общей структуре работы во многом подобны рассмотренному динамическому массиву. Аналогично, они динамически (вот сюрприз) расширяемы, они удаляют своё содержимое по выходу из области видимости. В дополнении, ассоциативные массивы также предоставляют итераторы для непрямого доступа к своим элементам.

Ассоциативный массив::где взять

Ассоциативный массив представлен в C++ шаблонными классами **map** и **unordered_map**, которые определены в **<map>** и **<unordered_map>** соответственно.

```
1 #include <map>
2 #include <unordered_map>
```

Общая форма для создания объектов типа *ассоциативный массив*:

```
1 map<KeyType, ValueType> var_name( args... );
```

, где **map** - название самого типа; **KeyType** - тип данных ключа; **ValueType** - тип данных значения, хранимого по ключу; **var_name** - имя переменной; **args...** - аргументы, передаваемые в конструктор объекта.
Ассоциативный массив стандартной библиотеки внутри хранит каждую пару «ключ-значение» как объект **pair<KeyType, ValueType>**. **<map>** и **<unordered_map>** различаются организацией хранения массива таких пар.

Ассоциативный массив::конструкторы

Далее расматриваются операции на примере типа <**unordered_map**>.

```
(1) unordered_map<KeyType, ValueType> my_hash()
(2) unordered_map<KeyType, ValueType>
my_hash(initializer_list<pair<KeyType, ValueType>>)
```

1. **(1)** – конструктор без параметров, просто создаёт ассоциативный массив, готовый для помещения элементов;
2. **(2)** – конструктор, позволяющий *инициализацию списком* путём передачи в него массива элементов **pair<KeyType, ValueType>**.

Ассоциативный массив::конструкторы

Примеры:

```
1 unordered_map<int, string> hash1;
2
3 // А ещё можно так:
4 unordered_map<int, string> hash2 = {
5                 { 25, "Строка 1"},
6                 { -8, "Что-то ещё"},
7                 { 42, "Make all humans better" },
8                 { 12, "И опять строка" }
9           };
```

Ассоциативный массив::итераторы

Обход всех элементов контейнера:

```
1  unordered_map<char, string> hash1 = {
2          {'a', "Feel"},
3          {'v', "Быть"},
4          {'z', "тому"},
5          {'%', "не быть"}
6      };
7
8  cout << '\n';
9
10 for (pair<char, string>& elem : hash1) {
11   cout << "Символ " << elem.first
12        << " означает " << elem.second
13        << endl;
14 }
```

В **10** строке может быть использовано ключевое слово **auto** вместо явного указания типа элемента.

Ассоциативный массив::методы контейнера

```
unordered_map<KeyType, ValueType> hash;
(1) size_t hash.size();
(2) size_t hash.max_size();
(3) bool hash.empty();
(4) void hash.clear();
```

- **(1)** - узнать текущий размер массива
- **(2)** - узнать потенциально максимальное количество элементов
- **(3)** - метод возвращает **true** если массив не содержит ни одного элемента, **false** - в противоположном случае
- **(4)** - удалить все элементы из массива

```
1 unordered_map<int, int> hash1 = { {1, 5}, {2, 6} };
2 cout << "\nРазмер хэша: " << hash1.size();
3 hash1.clear();
4 cout << "\nРазмер хэша: " << hash1.size();
```

Ассоциативный массив::добавление элементов

```
(1) hash[KeyType & key] = value;
(2) hash.insert(...);
(3) hash.emplace(args)
```

- **(1)** - получить ссылку на элемент для ключа **key**
- **(2)** - добавить элементы в ассоциативный массив. Аргументами, как правило, являются **пары значений** (слайд 53). Далее будут показаны на примерах.
- **(3)** - создать и добавить элемент в ассоциативной массив. **args** - аргументы, которые будут переданы конструктору объекта **pair<KeyType, ValueType>{args}**

Использование оператора «квадратные скобки»:

```
1 unordered_map<int, string> hash1 = { {1, "Всё good"} };
2
3 hash1[101] = "Другая строка";
4 cout << hash1[101];
```

Ассоциативный массив::добавление элементов

```
1  unordered_map<string, double> favor_courses;
2  pair<string, double> e8ty{"Электричество", 8.8};
3  // Неявно создаём пару значений string, double
4  // и добавляем её в ассоциативный массив
5  favor_courses.insert({"Тер. мех", 10.0});
6  // Копируем содержимое пары из строки 2
7  favor_courses.insert(e8ty);
8  // Можно передавать набор пар, аналогично
9  // слайду 61, пример, строка 4
10 favor_cources.insert({
11     {"УМФ", 9.2},
12     {"Тер. вер.", 8.7},
13     {"Языки программирования", 1.5}
14 });
15
16 for (const auto& elem : favor_courses) {
17   cout << "Предмет <<" << elem.first << ">>"
18        << " получает оценку: " << elem.second << endl;
19 }
```

Ассоциативный массив::добавление элементов

```
1 unordered_map<string, int> phrases;
2 phrases.emplace("Умная поговорка быть здесь должна",
3                 100);
4 phrases.emplace(make_pair("What's up?", 50));
5
6 for (const auto& elem : phrases) {
7   cout << "Фраза <<" << elem.first << ">>"
8        << " получила рейтинг: " << elem.second << endl;
9 }
```

Ассоциативный массив::доступ к элементам

```
(1) ValueType& hash[KeyType & key];
(2) ValueType& hash.at(KeyType & key);
```

- **(1)** - получить ссылку на элемент для ключа **key**
- **(2)** - получить ссылку на элемент для ключа **key**. Только для существующих элементов!

```
1  unordered_map<int, string> hash1 = { {1, "Feel good"} ↩
       };
2  hash1[22] = "Другая строка";
3  cout << hash1[1];
4  hash1.at(1) = "Снова и снова";
5  cout << hash1[1];
6  // Ключ не существует — создаём его с помощью
7  // конструктора без параметров (если определён)
8  cout << hash1[25];
9
10 try { cout << hash1.at(26) }
11 catch (out_of_range & ex ) { cout << ex.what(); }
```

Ассоциативный массив::выгнать элементы

```
(1) size_t hash.erase(const KeyType & key);
(2) size_t hash.erase(const iterator pos);
```

- **(1)** - удалить элемент для ключа **key**. Если удаление прошло удачно - возвращаемое значение равно **единице**, иначе - **нулю**
- **(2)** - удалить элемент, на который указывает итератор **pos**

```
1 unordered_map<char, string> hash1 = { {'a', "Feel"} };
2 hash1['*'] = "Другая строка";
3 hash1['@'] = "Третья строка";
4
5 hash1.erase('@');
6 cout << "\nРазмер хэша: " << hash1.size();
```

Ассоциативный массив::выгнать элементы

Ещё примеры:

```
1 unordered_map<char, string> hash1 = { {'a', "Feel"} };
2 hash1['*'] = "Другая строка";
3 hash1['@'] = "Третья строка";
4 cout << "\nРазмер хэша: " << hash1.size();
5
6 hash1.erase( hash1.begin() );
7 cout << "\nРазмер хэша: " << hash1.size();
```

но надо ли на практике?

Ассоциативный массив::про **map** не забываем

Все перечисленные примеры - работают и для типа **map** из библиотеки <**map**>:

```
1 map<string, int> phrases;
2 phrases.emplace("Умная поговорка быть здесь должна",
3                 100);
4 phrases.emplace(make_pair("What's up?", 50));
5
6 for (const auto& elem : phrases) {
7   cout << "Фраза <<" << elem.first << ">>"
8        << " получила рейтинг: " << elem.second << endl;
9 }
```

замена одного типа на другой никак не поменяла действия с самим объектом ассоциативного массива.
В тоже время, все примеры использовали либо *фундаментальные* типы, либо типы из стандартной библиотеки C++ (**string**). Разница между **map** и **unordered_map** возникает тогда, когда в качестве ключа захочется использовать пользовательские типы данных.

Ассоциативный массив::намёки на то, что скрыто

map: внутренняя структура.
Объекты типа **map** внутри себя хранят пары объектов, упорядоченные по ключу. По умолчанию упорядочение происходит с помощью оператора «меньше», т.е. внутри элементы отсортированы по возрастанию значений ключа. Как правило, внутри контейнера используется *древовидная* структура для хранения пар «ключ-значение». При инстанцировании шаблонного класса **map** можно передать тип функционального объекта, который будет сравнивать значения ключей ассоциативного массива.

Ассоциативный массив::намёки на то, что скрыто

Как сделать собственный тип ключём **map**

Для использования пользовательского типа в качестве ключа **map** существуют две возможности:

1. определить перегрузку оператора «меньше» для собственного типа;
2. определить тип функционального объекта, который будет сравнивать значения собственного типа.

Ассоциативный массив::намёки на то, что скрыто

Сама вставка элемента в ассоциативный массив происходит по следующему правилу:

Вставка элемента в объект типа **map**

Ключ добавляемого элемента сравнивается с уже имеющимися.

- если результат сравнения вернул **true**, то происходит переход к следующему ключу;
- если результат сравнения вернул **false**, происходит вставка на данное место внутри **map**.

Стоит отметить, что в зависимости от результата операции сравнения, один ключ может быть заменён другим и их значения могут не совпадать полностью.

Ассоциативный массив::свои типы для ключа

Продемонстрируем на примере. Пусть дана структура:

```
1 struct MyCoolKey
2 {
3   string name;
4   int rate;
5   double value;
6 };
```

Хотим сделать её ключом для **map**:

```
7 map<MyCoolKey, string> my_map;
```

Компилирование такого фрагмента приведёт к ошибке, поскольку по умолчанию для структур никаких операций сравнения не определяется.

Ассоциативный массив::свои типы для ключа

Для исправления ошибки компиляции перегрузим оператом «меньше»

```
1  bool operator<(const MyCoolKey& lhs,
2                 const MyCoolKey& rhs)
3  {
4    return lhs.rate < rhs.rate;
5  };
```

Теперь ошибки компиляции пропадут:

```
6  map<MyCoolKey, string> my_map;
7
8  my_map[{"1st", 5, 2.54}] = "Первое значение";
9  my_map[{"2end", 15, -1.74}] = "Второе";
10 my_map[{"3rd", 28, 0.58}] = "Третье";
```

Из-за того, что ключём является значение структуры, в квадратных скобках используются фигурные скобки для указания полей конкретных объектов. **Фактически**, вставка происходит по полю типа **int**.

Ассоциативный массив::свои типы для ключа

Если попробуем вставить два ключа с одинаковым значением поля **rate**, в ассоциативный массив будет сохранено только последнее значение:

```
6  map<MyCoolKey, string> my_map;
7
8  my_map[{"1st", 5, 2.54}] = "Первое значение";
9  my_map[{"2end", 15, -1.74}] = "Второе";
10 my_map[{"3rd", 28, 0.58}] = "Третье";
11 my_map[{"4th", 15, 12.99}] = "Уже не Второе";
12
13 cout << "Размер my_map: " << my_map.size() << endl;
14 for (const auto& elem : my_map) {
15   cout << "{" << elem.first.name << ", "
16        << elem.first.rate << "} => "
17        << elem.second << endl;
18 }
```

В результате **my_map** будет хранить только три елемента.

Ассоциативный массив::пример

Для примера с функциональным объектом покажем такую задачку: пусть есть набор температур и среднее значение некой физической величины в каждой точке. Показать все значения в порядке убывания температуры.
На практике, подобная задача может возникнуть при расчёте различных средних значений температурной зависимости некоторой величины. Как правило, значения для расчёта считываются из файла. При этом, иногда разумно каждое значение температуры хранить как строку для того, чтобы не иметь проблем с округлением и последующим выводом результата на экран/в файл.

Ассоциативный массив::пример

Итак, для решения выше поставленной задачи можно использовать типа **map**. Для хранения пар «температура» → «среднее значение» подходит следующий тип:

```
1 map<string, double> temp_values;
```

Но по умолчанию ключи будут сравниваться как строки (лексикографическое сравнение). Для того, чтобы внутри ассоциативного массива хранились в порядке убывания их значений, заведёт класс функционального объекта, который будет сравнивать строки как значения типа **double**

```
2 class MyComparator
3 {
4 public:
5   bool operator()(string& lhs, string& rhs)
6   {
7     return stod(lhs) > stod(rhs);
8   }
9 };
```

Ассоциативный массив::пример

Компаратор определён, осталось передать его в шаблонный класс **map**. Для этого используется третий параметр шаблона. В примерах ранее он был определён по умолчанию.

```
1  map<string, double, MyComparator> temp_values = {
2    {"1.1", 0.23},   {"0.75", 0.88},
3    {"1.345", 0.11}, {"1.65", 0.75},
4    {"1.89", 0.45},  {"0.5", 0.058}
5  };
6  temp_values["2.5"] = 0.98;
7
8  for (const auto& elem : temp_values) {
9    cout << elem.first << " : "
10        << elem.second << endl;
11 }
```

В результате, значения будут выведены в порядке убывания по температуре. Конечно, подобную задачу можно решить через динамические массивы и сортировку, но в данном случае в явном виде не происходит ничего, кроме вставки нужных значений.

Ассоциативный массив::намёки на то, что скрыто

Объекты шаблонного типа **unordered_map** внутри себя хранят пары объектов неупорядоченно. Он спроектирован для быстрого доступа по ключу к любому хранимому значению. Для этого от каждого добавляемого ключа вычисляется **хеш-функция** и с помощью полученного значения происходит сохранение пары «ключ-значение» во внутренней структуре конкретного объекта.
Поэтому ключём ассоциативного массива **unordered_map** может быть любой тип данных, для объектов которого возможно вычисление хеша.

Ассоциативный массив::намёки на то, что скрыто

Суть **хеш-функции** уже обсуждалась. Для повторения, хеш-функция при помощи некоторого алгоритма преобразует произвольное значение в значение фиксированной длины. В стандартной библиотеке C++ в качестве значения хэш-функции был выбран тип **size_t** (целое беззнаковое число, как правило, 8-байтовое). Для вычисления хеша от *фундаментальных типов* в заголовочном файле <**functional**> определён шаблонный класс **hash**, который служит для создания функиональных объектов, которые возвращают значение хеш-функции для конкретных значений различных типов. Для одинаковых значений вычисленный хеш будет также совпадать.

Ассоциативный массив::намёки на то, что скрыто

Прототип класса **hash** схематично можно представить как:

```
1  template<typename T>
2  class hash
3  {
4  public:
5    size_t operator()(const T& value)
6    {
7      // реализация алгоритма хеширования
8      return значение_хеша;
9    }
10 };
```

Как видно, типичный пример функционального объекта, зависящего от параметра.

Ассоциативный массив::намёки на то, что скрыто

Несколько примеров на **hash**:

```
1  #include <functional>
2
3  hash<char*> ptr_hasher;
4  hash<string> str_hasher;
5
6  const char* c_str1 = "Проверка",
7            * c_str2 = "Проверка";
8  string str1{c_str1}, str2{c_str2};
9
10 cout << "Совпадают ли значения хэшей?" << boolalpha
11      << endl;
12
13 cout << "От c_str1 и c_str2: "
14      << (ptr_hasher(c_str1) == ptr_hasher(c_str2)) << ↩
            endl;
15
16 cout << "От str1 и str2: "
17      << (str_hasher(str1) == str_hasher(str2)) << endl;
```

Ассоциативный массив::намёки на то, что скрыто

В примере предыдущего слайда в первом случае сравниваются не значения строк, а значения **указателей**. Значения адресов различны для двух разных переменных, поэтому будет выведено **false**. Для объектов класса **string** хеш вычисляется от символов строки, поэтому для разных строк будет вычислено одно и тоже значение. Дополнительные примеры на фундаментальные типы:

```
1 #include <functional>
2
3 hash<int> int_hasher;
4 cout << "hash(-255) = " << int_hasher(-255) << endl;
5 cout << "hash(1788) = " << int_hasher(1788) << endl;
6
7 hash<double> real_hasher;
8 cout << "hash(0.75) = " << real_hasher(0.75) << endl;
```

Ассоциативный массив::намёки на то, что скрыто

Тип **hash** напрямую связан с ассоциативным массивом **unordered_map**: он используется по умолчанию для вычисления хешей от значений ключа. На практике это означает, что когда передаётся тип ключа шаблонным параметром, компилятор проверяет, можно ли для данного типа создать объект **hash<Type>** для последующего вычисления хешей.

Как сделать собственный тип ключём **unordered_map**

Аналогично **map**, есть две возможности:

1. специализировать шаблонный класс **hash** из **<functional>** для собственного типа;
2. передать третьим шаблонным параметром тип, который представляет собой функциональный объект для вычисления хеша от значений собственного типа.

Ассоциативный массив::ключ для **unordered_map**

Специализация типа **hash**.
Для примера используем структуру **MyCoolKey**. Для вычисления хеша от структур применим такой алгоритм: вычисляем хеш от каждого значения и объединяем их логическим «или». Для специализации шаблонного класса под собственную структуру, эта структура должна иметь перегруженную версию оператора ==:

```
1 bool operator==(const MyCoolKey& lhs,
2                 const MyCoolKey& rhs)
3 {
4   return (lhs.name == rhs.name)
5       && (lhs.rate == rhs.rate)
6       && (lhs.value == rhs.value);
7 };
```

Ассоциативный массив::ключ для **unordered_map**

Теперь специализацию можно определить так:

```
1  namespace std { // Специализация долна быть в том же
2  template<>      // пространстве имён, что и сам класс.
3  class hash<MyCoolKey>
4  {
5  public:
6      using result_type = size_t;
7      using argument_type = MyCoolKey;
8
9      size_t operator()(const argument_type& key) const
10     { // указание метода как *const* — обязательно!
11       using namespace std;
12       size_t str_val = hash<string>{}(key.name),
13              int_val = hash<int>{}(key.rate),
14              real_val = hash<int>{}(key.value);
15       return str_val | int_val | real_val;
16     }
17 };
18 } // Поэтому оборачивание в std — обязательно
```

Ассоциативный массив::ключ для **unordered_map**

В примере выше конструкция **hash<string>{}** означает создание временного объекта типа **hash<string>**, у которого затем сразу вызывается перегруженный оператор «круглые скобки». Теперь собственную структуру можно использовать в качестве ключа:

```
1  unordered_map<MyCoolKey, string> fast_map;
2
3  fast_map[{"1st", 5, 2.54}] = "Первое значение";
4  fast_map[{"2end", 15, -1.74}] = "Второе";
5  fast_map[{"3rd", 28, 0.58}] = "Третье";
6  fast_map[{"4th", 15, 12.99}] = "Уже ничто не затрёт";
7
8  cout << "Размер fast_map: " << fast_map.size() << endl;
9  for (const auto& elem : fast_map) {
10   cout << "{" << elem.first.name << ", "
11        << elem.first.rate << "} => "
12        << elem.second << endl;
13 }
```

Ассоциативный массив::ключ для **unordered_map**

unordered_map: собственный функциональный объект для хэширования. Для второго способа определяется собственный класс:

```
1  class CoolKeyHash
2  {
3  public:
4      size_t operator()(const MyCoolKey& key) const
5      { // указание метода как *const* — обязательно!
6        using namespace std;
7        size_t str_val = hash<string>{}(key.name),
8               int_val = hash<int>{}(key.rate),
9               real_val = hash<int>{}(key.value);
10       return str_val | int_val | real_val;
11     }
12 };
```

Ассоциативный массив::ключ для **unordered_map**

И этот класс передаётся третьим параметром шаблона:

```
1  unordered_map<MyCoolKey, string, CoolKeyHash> other_mp;
2
3  other_mp[{"1st", 5, 2.54}] = "Первое значение";
4  other_mp[{"2end", 15, -1.74}] = "Второе";
5  other_mp[{"3rd", 28, 0.58}] = "Третье";
6  other_mp[{"Нечто", 15, 12.99}] = "Пример есть пример";
7
8  cout << "Размер other_mp: " << other_mp.size()
9       << endl;
10 for (const auto& elem : other_mp) {
11   cout << "{" << elem.first.name << ", "
12        << elem.first.rate << "} => "
13        << elem.second << endl;
14 }
```

Ассоциативный массив::заметки на полях

Пара моментов, касающихся **unordered_map**:

- типы **unordered_map<MyCoolKey, string>** и **unordered_map<MyCoolKey, string, CoolKeyHash>** хоть и делают одно и тоже, с точки зрения компилятора являются различными: переменным одного из них никак не присвоить значение переменных другого;
- когда нужен ассоциативный массив и не нужен особый порядок перебора значений, предпочитайте использовать **unordered_map** вместо **map**.

Контейнеры::множество

В стандартной библиотеке C++ доступен ещё один контейнер, представляющий собой *множество* – набор уникальных значений.
Для этого определены два заголовочных файла: <**set**> и <**unordered_set**>. Смысл разделения аналогичен ассоциативным массивам: в первом случае элементы множества хранятся в упорядоченной структуре (используется сравнение «меньше» по умолчанию), во втором - от каждого элемента вычисляется хэш и сохраняется внутри контейнера.
Типы множеств являются шаблонными; передать тип-сравнение или собственную тип для хэширования можно вторым параметром шаблона.
Далее рассматриваем только случай <**unordered_set**>.

Множество::какие варианты

<**unordered_set**> предоставляет 2 класса: **unordered_set** и **unordered_multiset**. Первый определяет множество, в котором каждое значение представлено только единожды. В случае второго - для каждого элемента вычисляется, сколько раз он был добавлен в множество.

Множество::методы

Методы для работы с множествами во многом аналогичны уже рассмотренным в других контейнерах:

- **insert** / **emplace** - добавляют элемент в множество;
- **erase** - удаляет элемент;
- **find** - получть итератор на элемент, если он присутствует в множестве, либо специальное значение **end** при отсутствии;
- **count** - проверить количество добавления элемента в множество. Для **unordered_set** этот метод всегда возвращает или **1**, или **0**. Для **unordered_multiset** - возвращает сколько раз элемент был добавлен в множество.

Подробно интерфейс классов смотреть тут:
http://www.cplusplus.com/reference/unordered_set/
http://www.cplusplus.com/reference/set/

Множество::сразу к примеру

Задачка: читать из файла слова и вывести на экран только встречающиеся в заданном множестве.

```
1  unordered_set<string> magic_words = {
2      "интерфейс", "наследование", "итератор",
3      "метод", "инкапсуляция", "ссылка"
4  };
5  magic_words.insert("литерал");
6  magic_words.insert("указатель");
7
8  ifstream input_file{"my_text.txt"};
9
10 if (input_file) {
11   string word;
12   while (input_file >> word) {
13     if (magic_words.count(word) != 0) {
14       cout << "Слово \"" << word << "\" "
15            << "найдено в тексте\n";
16     }
17   }
18 }
```

Множество::multiset

Пример на **multiset** узнать количество вхождений каждого слова в данный текст.

```
1  unordered_multiset<string> my_words = {
2      "интерфейс", "наследование", "итератор",
3      "метод", "инкапсуляция"
4  };
5  my_words.insert("литерал");
6  my_words.insert("указатель");
7  // Удаляем элемент из множества
8  my_words.erase("ссылка");
9
10 ifstream input_file{"my_text.txt"};
11 if (input_file) {
12   string word;
13   while (input_file >> word) {
14     if (my_words.count() != 0) {
15       my_words.insert(word);
16     }
17   }
18 }
```

Важная деталь об **multiset**

Оба типа для множеств предоставляют итераторы для обхода контейнера. Как было проверено на практике, при обходе объекта **multiset** итератор будет проходить каждый элемент столько раз, сколько этот элемент был добавлен в объект (вызван метод **insert**). Но можно воспользоваться следующим знанием: элементы в **multiset** с одинаковым значением будут обходиться итератором последовательно.

Продолжаем пример: узнать количество вхождений каждого слова в данный текст.

```
19 for (auto it = my_words.begin();it != my_words.end()) {
20   const auto word_count = my_words.count(word);
21   cout << "Слово <" << word << "> присутствовало"
22        << "в тексте "
23        << word_count
24        << " раз" << endl;
25
26   // Функция advance из <iterator> сдвигает переданный
27   // итератор (*it*) на число позиций, переданное
28   // вторым аргументом (*word_count*). Фактически, она
29   // делает операцию: it = it + word_count
30   advance(it, word_count);
31 }
```