Лекция XIII

Автоматический вывод типа переменной компилятором, или когда хочется набирать меньше символов

auto для вывода типов

Компилятор C++ при анализе исходного кода оладает полной информациях о типах переменных/значений программы. Во многих случаях для него не проблема вывести тип переменной самостоятельно, без его явного указания. Но для этого переменной нужно обязательно **присвоить значение**. А вместо **типа данных** написать ключевое слово **auto**

```
1  auto int_var = 15;
2  // int_var становится переменной типа int
3
4  auto real_var = 15.6;
5  // int_var – переменной типа double
6
7  auto str = "Какая-то строка";
8  // str получает тип const char *
9
10 auto str_obj = std::string{"Какая-то строка-2"};
11 // str получает тип std::string
```

auto для вывода типов

Но для простых типов данных применение **auto** при определении переменных **строго не рекомендуется**. Когда же есть польза?

Первый пример: **for-range**

```
1  DynArray1D my_arr{5};
2  my_arr << 1.2 << 1.3 << 1.4 << 1.5 << 1.6;
3
4  // Ранее в примерах:
5  for (double& elem : my_arr) {
6    elem *= elem;
7  }
8
9  // С auto можно делать так:
10 for (auto& elem : my_arr) {
11   elem *= elem;
12 }
13 // Или так (получение копии элементов):
14 for (auto elem : my_arr) {
15   cout << elem << ", ";
16 }
```

auto для вывода типов

Второй пример: динамическое создание объектов класса

```
1 // Где-то раннее:
2 DynArray1D *p_arr2 = new DynArray1D{8, 555.555};
3
4 // С auto можно устранить дублирование названия типа:
5 auto p_arr2 = new DynArray1D{8, 555.555};
```

Оператор **new** возвращает указатель на типа **DynArray1D**, этот же тип получает переменная **p_arr2**.

auto для вывода типов

Третий пример: получение итераторов

```
1  DynArray1D da{20, 0.0};
2  for (auto& elem : da) {
3    elem = Rand::get_a_b(-50; 50);
4  }
5
6  // Сортируем первую половину массива
7  auto sort_stop_it = end(da) - da.length() / 2;
8  std::sort(begin(da), sort_stop_it);
9
10 cout << da << endl;
```

Нет неоходимости вспоминать, что в случае класса **DynArray1D** в качестве *итераторов* используются указатели на **double**.

Дополнительная порция информации про шаблоны

В шаблонные функции или типы можно передать произвольное число аргументов шаблона. Для этого используется специальный синтаксис для шаблонного параметра:

```
1 template<TArgs...>
2 void fn(TArgs&... values);
```

Здесь троеточие это специальный символ как раз и означающий, что в параметре **TArgs** содержатся все реально переданные (при инстанцировании шаблона) типы.
Технический термин - **parameter pack** (или template parameter pack).
В примере, во второй строке фактически объявлена функция, в которую передаются значения разных типов (каждое значение передаётся *по ссылке*). Эти значения упакованы в переменную **values**.

В шаблонные функции или типы можно передать произвольное число аргументов шаблона. Для этого используется специальный синтаксис для шаблонного параметра:

```
1 template<TArgs...>
2 void fn(TArgs&... values);
```

Здесь троеточие это специальный символ как раз и означающий, что в параметре **TArgs** содержатся все реально переданные (при инстанцировании шаблона) типы.
Технический термин - **parameter pack** (или template parameter pack).
В примере, во второй строке объявлена функция, в которую передаются значения разных типов (каждое значение передаётся *по ссылке*). Эти значения упакованы в переменную **values**.

Из практических примеров, где полезны аргументы шаблона произвольной длины, рассмотрим реализацию **print** из **ffhelpers.h**. Эта функция позволяла печатать практически любое значение любого типа данных и принимала любое число аргументов.
Её работа была основана на рекурсии и произвольном числе аргументов шаблона. Идея такая:

- создаём шаблонную функцию с двумя шаблонными параметрами: 1) тип значения для вывода печати в текущий момент; 2) остальные типы упаковынные в спец. аргументе;
- напечатав нужное значение, сделаем рекурсивный вызов функции для оставшихся значений;
- специализируем функцию на случай, когда осталось только одно значение произвольного типа.

Чтобы улучшить **print** из **ffhelpers.h** сделаем возможность печати в любой поток ввода-вывода C++. Для закрепления одной темы с классами, сделаем это собственный тип. Заодно увидим, что шаблонными могут быть методы вполне себе конкретного класса.

В классе будет только одно поле - ссылка на поток вывода:

```
1 class Printer
2 {
3 public:
4   ...
5
6 private:
7   std::ostream& _stream_ref;
8 };
```

Для её инициализации будет определён конструктор:

```
1  #include <ostream>
2
3  class Printer
4  {
5  public:
6    Printer(std::ostream& out_stream) : _stream_ref{↩
         out_stream}
7    {}
8
9  private:
10   std::ostream& _stream_ref;
11 };
```

Добавим методы для печати чего угодно:

```
1  class Printer
2  {
3  public:
4    Printer(std::ostream& out_stream) : _stream_ref{↩
         out_stream}
5    {}
6    // Все аргументы передаём по константным ссылкам
7    template<typename T, typename... TOther>
8    void print(const T& value,
9               const TOther&... other_args);
10
11   // Данная функция нужна для завершения рекурсии,
12   // когда останется только одно значение для печати.
13   template<typename T>
14   void print(const T& value);
15
16 private:
17   std::ostream& _stream_ref;
18 };
```

Реализуем методы:

```
1  template<typename T, typename... TOther>
2  void Printer::print(const T& value,
3                      const TOther&... other_args)
4  {
5    _stream_ref << value;
6    // Здесь троеточие означает, что происходит ↩
       распаковка аргументов в отдельные значения
7    print(other_args...);
8  }
9
10
11 template<typename T>
12 void Printer::print(const T& value)
13 {
14   _stream_ref << value;
15 }
```

Данные методы будут печатать объект любого типа, для которого **переопределён** оператор вывода.

И появляется возможность уйти от кучи операторов «

```
1  Printer std_out{std::cout};
2  string str = "строка класса string"
3  std_out.print(123, " ", 5.634, " --- ", str, "\n");
4
5  DynArray1D arr;
6  arr << 1.2 << 1.3 << 1.4 << 1.5 << 1.6;
7  std_out.print("Массив: ", arr, "\n");
8
9  // Вывод в файл
10 ofstream out{"myfile.txt"};
11 if (out) {
12   Printer file_out{out};
13   file_out.print("Массив: ", arr, "\n");
14 }
```

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. Стоит отметить, что все эти контейнеры представляют собой *шаблонные классы*. Что означает, что они могут работать как с **фундаментальными**, так и с **пользовательскими** типами.

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.

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. Память под элементы не выделяется.
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**.

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);
```

Можно заметить, что для вызова конструктора в примерах делается через круглые скобки. Хотя при расмотрении основ создания собственных классов был совет предпочитать *фигурные скобки*.

Контейнеры. Динамический массив

Дело в том, что для динамического массива **vector** определён ещё один специальный конструктор, позволяющий инициализировать его объект значениями в момент создания:

```
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};
```

Иногда такой способ создания объектов массива предпочтителен. И 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 представленного шаблонным типом **vector** определена простая рекомендация:

Круглые скобки рулят

Для разрешения необнозначности при вызове конструкторов класса **vector** (слайд 20) предпочитайте ***круглые скобки***.

Кроме того, стоит заметить 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};
```

Динамический массив: методы для работы с количеством элементов

```
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 возвращает **true** если массив не содержит ни одного элемента, **false** - в противоположном случае.

Контейнеры. Динамический массив

Динамический массив: методы для работы с количеством элементов

```
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: методы для работы с количеством элементов, примеры

```
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: методы для доступа 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: методы для доступа 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: методы для доступа к элементам

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

На примере массива со слайда 26:

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

Динамический массив: методы для доступа 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();
```

Контейнеры. Динамический массив

Динамический массив: работа с итераторами

```
(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: работа с итераторами

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++.

Контейнеры. Динамический массив

Динамический массив: работа с итераторами

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()**

Контейнеры. Динамический массив

Динамический массив: работа с итераторами

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/

Контейнеры. Динамический массив

Динамический массив: методы для добавления 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: методы для добавления 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;
```

Контейнеры. Динамический массив

Динамический массив: методы для добавления 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;
```

Контейнеры. Динамический массив

Динамический массив: методы для добавления 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 };
```

Контейнеры. Динамический массив

Динамический массив: методы для добавления 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.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.

Контейнеры. Динамический массив

Динамический массив: методы для удаления конкретных элементов массива

```
(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: методы для удаления конкретных элементов массива

```
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;
```

Контейнеры. Динамический массив

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:
13 int2D[5][4] = 888;
14 cout << "Что-нибудь: " << int2D[20][102] << endl;
```

Динамический массив: 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 }
```

Контейнеры. Статический массив

Статический массив: стандартная библиотека C++ предоставляет шаблонный класс **array** для создания и манипуляции массивами **неизменяемой** длины как объектами (а не уже известными встроенными типами). 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 только **конструктор без параметров**.

Контейнеры. Статический массив

Статический массив: стандартная библиотека C++ предоставляет шаблонный класс **array** для создания и манипуляции массивами **неизменяемой** длины как объектами (а не уже известными встроенными типами). 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 (см. слайды 21-22).

Контейнеры. Статический массив

Статический массив: для доступа к элементам предоставляются теже возможности, что для 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;
```

Контейнеры. Статический массив

Статический массив: запрос длины и проверка на пустоту:

```
(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;
```

Контейнеры. Статический массив

Статический массив: аналогино динамическому массиву, для шаблонного класса **array** 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;
```

Контейнеры. Статический массив

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.

Контейнеры. Статический массив

Статический массив: опять же, аналогино динамическому массиву, класс **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');
```

Контейнеры. Пара значений

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;
```

Контейнеры. Пара значений

Пара значений: создание с помощью шаблонной функции **make_pair**, которая 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;
```

Контейнеры. Ассоциативный массив

Ассоциативный массив - специальный тип данных, в котором индексом массива может быть объект произвольного типа. Известен также по терминам **«хеш»** и **«мап»** 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 шабланные классы, которые позволяют задать разные типы для ключа и значения.

Контейнеры. Ассоциативный массив

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 задания объектов данного класса есть:

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

, где **KeyType** - тип данных ключа, **ValueType** - тип данных второго значения, **var_name** - имя переменной, **args...** - аргументы, передаваемые в конструктор.
Оба типа внутри хранят каждую пару «ключ-значение» как объект **pair<KeyType, ValueType>** (слайд 53) и различаются организацией хранения массива таких пар.

Ассоциативный массив: конструкторы

`(1) unordered_map<KeyType, ValueType> my_hash()`

- **(1)** - конструктор без параметров, просто создаёт ассоциативный массив, готовый для помещения элементов. Стоит отметить, что каждый элемент представляет собой объект структуры **pair<KeyType, ValueType>**.

```
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, "Kill all humans" },
8                 { 12, "И опять строка" }
9           };
```

Контейнеры. Ассоциативный массив

Ассоциативный массив: методы для работы с количеством элементов

```
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();
```

Контейнеры. Ассоциативный массив

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 goo"} };
2
3  hash1[22] = "Другая строка";
4  cout << hash1[1];
5
6  hash1.at(1) = "Снова и снова";
7  cout << hash1[1];
8  // Ключ не существует — создаём его, если возможно
9  cout << hash1[25];
10
11 try { cout << hash1.at(26) }
12 catch (out_of_range & ex ) { cout << ex.what(); }
```

Ассоциативный массив: доступ к элементам

```
(3) size_t hash.erase[const KeyType & key];
```

- **(3)** - удалить элемент для ключа **key**. Если удаление прошло удачно - возвращаемое значение равно **единице**, иначе - **нулю**

```
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: обход всех елементов

```
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        << '\n';
14 }
```