# 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 }
```

Данная реализация не лишена недостатков, кому интересно - спрашивайте в прямом эфире.

# Стандартная библиотека C++. Контейнеры

Какие типы предоставляет язык для хранения данных.

# 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(unsigned count)
6
7 // (3)
8 vector<Type> var2(unsigned 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**.

# Контейнеры. Динамический массив

**Динамический массив**: основные конструкторы, примеры:

```
1 vector<int> int_array;
2
3 vector<double> real_array(10);
4
5 string base_value = "ABC";
6 vector<string> str_array(5, base_value);
7
8 // Можно делать и так:
9 vector<int> int_arr2 = {1, 5, 6, 7, 8, 10};
```

# Контейнеры. Динамический массив

**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. **(1)** - узнать текущий размер массива
2. **(2)** - узнать потенциально максимальное количество элементов
3. **(3)** - метод возращает **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();
```

# Контейнеры. Динамический массив

**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 за номером **n**
2. **(2)** - получить ссылку на элемент за номером **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  int_arr.at(3) = 14;
6
7  cout << "\nЧетвёртый: " << int_arr.at(3);
8  // Поведение неопределено:
9  cout << "\nНеизвестный: " << int_arr[3001];
10
11 // Здесь скрыта разница между (1) и (2)
12 try { cout << "\nНеизвестный: " << int_arr.at(3001); }
13 catch (std::out_of_range & ex) { cout << ex.what(); }
```

**Динамический массив**: методы для доступа к элементам

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

3. **(3)** - получить ссылку на первый элемент
4. **(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**: методы для добавления

```
(5) void my_arr.push_back(Type & value );
(6) template<typename... Args>
    Iterator my_arr.emplace(iterator position, Args
```

5. **(5)** - добавить элемент **value** в конец массива
6. **(6)** - удалить последний элемент

```
1 vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3 int_arr.push_back(888);
4 int_arr.push_back(777);
5 cout << "Последний элемент: " << int_arr.back();
6
7 int_arr.pop_back();
8 cout << "Последний элемент: " << int_arr.back();
```

**Динамический массив**: методы для добавления

```
(5) Type & my_arr.push_back(Type & value );
(6) Type & my_arr.pop_back();
```

5. **(5)** - добавить элемент **value** в конец массива
6. **(6)** - удалить последний элемент

```
1 vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3 int_arr.push_back(888);
4 int_arr.push_back(777);
5 cout << "Последний элемент: " << int_arr.back();
6
7 int_arr.pop_back();
8 cout << "Последний элемент: " << int_arr.back();
```

# Контейнеры. Очередь

```
1  #include <queue>
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    my_queue.push(num);
10 } while (num != 0);
11
12 cout << "Введённая очередь:\n";
13 while ( !my_queue.empty() ) {
14   cout << my_queue.front() << ' ';
15   my_queue.pop();
16 }
```

# Контейнеры. Стек

```
1  #include <stack>
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    my_stack.push(num);
10 } while (num != 0);
11
12 cout << "Введённый стек:\n";
13 while ( !my_stack.empty() ) {
14   cout << my_stack.top() << ' ';
15   my_stack.pop();
16 }
```

# 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. **(1)** - конструктор без параметров, просто создаёт экземпляр структуры с двумя полями, не присваивая никаких начальных значений созданному объекту
2. **(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, pair2;
2  pair1 = std::make_pair(555, 0.783);
3
4  cout << "\nЗначения pair1: " << pair1.first
5                               << ' '
6                               << pair1.second;
7
8  // Тоже будет работать – неявное преобразование
9  pair2 = std::make_pair('c', '&');
10 cout << "\nЗначения pair2: " << pair2.first
11                              << ' '
12                              << pair2.second;
```

# Контейнеры. Ассоциативный массив

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

```
1 #include <unordered_map>
```

Общая форма для задания объектов данного класса есть:

```
1 #include <unordered_map>
2
3 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.

**Ассоциативный массив**: конструкторы

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

1. **(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. **(1)** - узнать текущий размер массива
2. **(2)** - узнать потенциально максимальное количество элементов
3. **(3)** - метод возвращает **true** если массив не содержит ни одного элемента, **false** - в противоположном случае
4. **(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. **(1)** - получить ссылку на элемент для ключа **key**
2. **(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. **(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 }
```