Лекция V

10 ноября 2017

Пользовательские типы данных

Производные типы данных

Псевдонимы (aliases)

typedef
using (C++)

Составные типы данных

struct (в данной лекции)
class (в данной лекции)
enum (в другой раз)
enum class (C++, в другой раз)
union (не рассматриваем)

Пользовательские типы данных. Псевдонимы

Наследие C: оператор **typedef** - добавление псевдонимов для любого существующего типа данных

```
typedef <тип_данных> <псевдоним1>
                [, <пс2>, <пс3>, ...];
```

Пользовательские типы данных. Псевдонимы

C/C++: оператор **typedef**

```
1  typedef unsigned long long ullint_t;
2
3  ullint_t              val1 = 444;
4  unsigned long long    val2 = val1;
5
6  // Объявляем псевдонимы для типа double
7  // и указателя на double
8  typedef double amperage_t, *amperage_ptr;
9
10 amperage_t val1 = 5.55;
11 // Два варианта определения переменных-↩
       указателей
12 amperage_ptr p_amp1 = &val1;
13 amperage_t *p_amp2 = p_amp1;
```

Пользовательские типы данных. Псевдонимы

Современный C++: оператор **typedef**

Полезный совет

При написании программ на C++ не используйте **typedef**, если компилятор поддерживает стандарт C++11 и новее

Решение

Используйте **using**

Пользовательские типы данных. Псевдонимы

C++11: **using** - добавление псевдонимов, аналогичное **typedef**, возможно с более понятным синтаксисом

```
using <псевдоним> = <тип_данных>;
```

Пользовательские типы данных. Псевдонимы

C++11: оператор **using**

```
1  const size_t TEN = 10;
2
3  /*
4  typedef unsigned long long ullint_t,
5                             *ullint_ptr,
6                             ullint_tenth_t[TEN];
7  */
8
9  using ullint_t       = unsigned int;
10 using ullint_ptr     = unsigned int*;
11 using ullint_tenth_t = unsigned int[TEN]
12
13 ullint_t val1 = 555;
14 ullint_ptr ptr1 = &val1;
15 cout << *ptr1;
```

Пользовательские типы данных. Псевдонимы

Предметно-ориентированные функции

```
1 using temperature_t   = double;
2 using temperature_ptr = double*;
3
4 temperature_t find_start_point();
5 void make_dynamic(temperature_t start,
6                   temperature_ptr end,
7                   temperature_ptr step);
```

Составные типы данных. Структуры

Структура (в смысле языка C) - это составной тип данных, объединяющий множество *проименнованных* типизированных элементов. **Элементы структуры** называют **её полями**. Тип поля может быть любой, известный к моменту объявления структуры. Общий синтаксис:

```
struct [<название_структуры>]
{
  <тип_1> <поле_1_1> [, <поле_1_2>, ...]
  <тип_2> <поле_2_1> [, <поле_2_2>, ...]
  ...
  <тип_n> <поле_n_1> [, <поле_n_2>, ...]
} [переменная1, переменная2, ...];
```

Использование структур: объявление переменных

```
1 struct MaterialPoint
2 {
3   int x, y;
4   int z;
5   double weight;
6 };
7
8 // начальные значения полей не ↩
    устанавливаются
9 MaterialPoint mp1, mp2, mp3;
```

Переменным структуры выделяются блоки памяти, каждый из которых состоит из трёх подблоков размера **int** и одного подблока размера **double**.

Использование структур: определение переменных

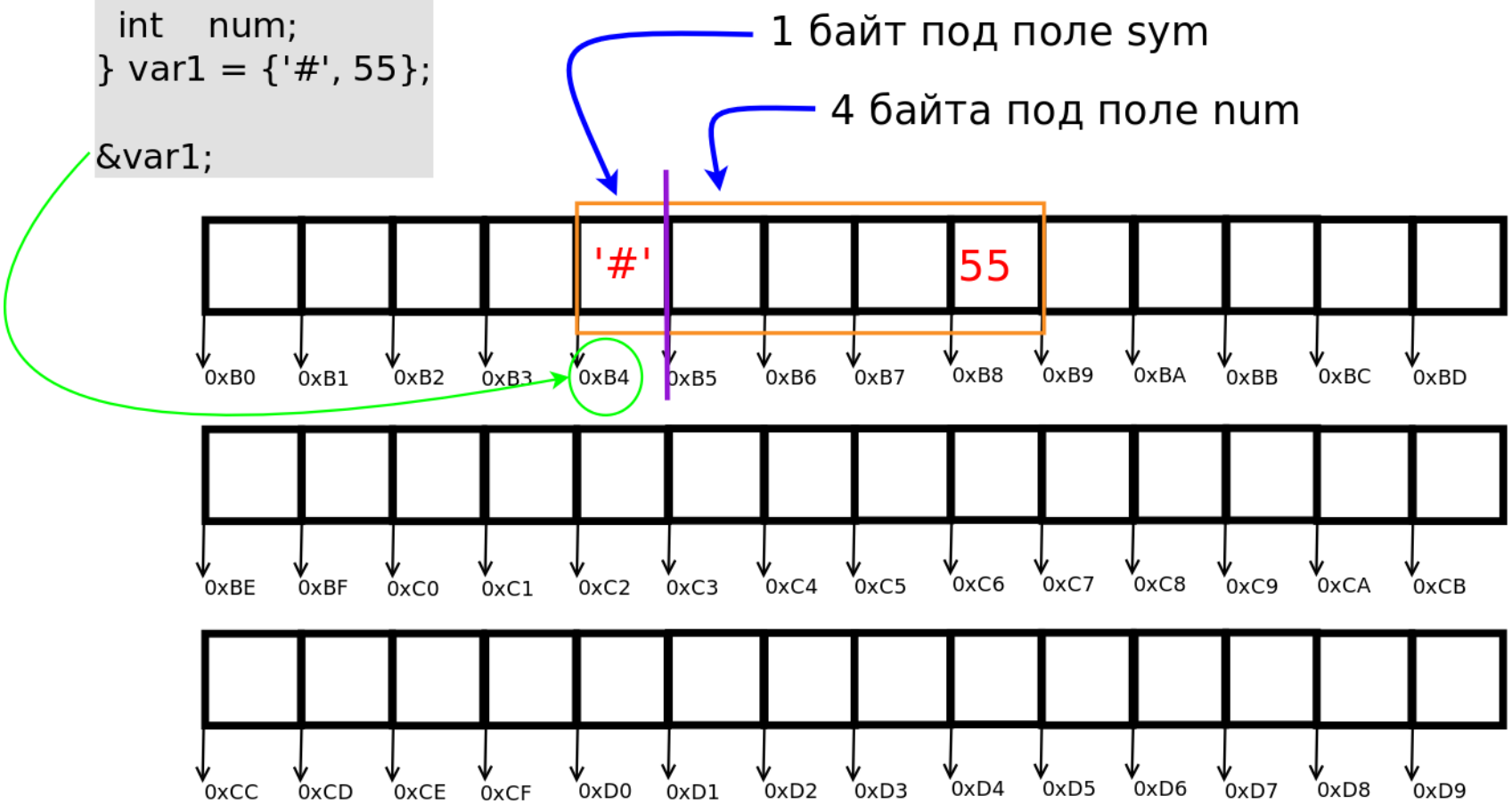
```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double weight;
5  };
6
7  // Используется список инициализаторов
8  MaterialPoint mp1 = { 3, 4, 5, 8.8 };
9  cout << mp1.weight << '\n';
10
11 MaterialPoint mp2 = { 5, 8 };
12 // mp2.z == 0, mp2.weigth == 0.0
13 cout << (mp2.z == 0) << ', ' << mp2.z;
```

При неполной инициализации поля, которым не поставлены никакие значения, получают нулевые значения.

Структуры: расположение переменной структуры в памяти

```
struct
{
  char sym;
  int    num;
} var1 = {'#', 55};

&var1;
```



Использование структур: обращение к полям переменных через оператор «.»

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double weight;
5  };
6
7  MaterialPoint mp;
8  mp.x = mp.y = 4;
9  mp.weight = 45.5;
10
11 cout << "Вес точки: " << mp.weight << "\n";
```

Составные типы данных. Структуры

Использование структур: определение структуры и объявление переменных одновременно (+ анонимная структура)

```
1  struct
2  {
3    int state, rank;
4    char name[80];
5    double factor;
6  } g_st1;
7
8  cout << "Введите имя: ";
9  cin.getline(g_st1.name, 80);
10
11 g_st1.state = 5;
12 g_st1.rank = -4;
13 g_st1.factor = 5.89;
```

Составные типы данных. Структуры

Использование структур: указатели на переменную структуры и оператор «->»

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double weight;
5  };
6
7  MaterialPoint mp1 = { 3, 4, 5, 8.8 };
8  MaterialPoint *p_mp = &mp1;
9
10 cout << (*p_mp).y << '\n';
11
12 // Получение значения поля по указателю
13 cout << p_mp->weight << '\n';
```

Использование структур: параметры функции

```
1  struct MaterialPoint
2  {
3    int x, y;
4    int z;
5    double weight;
6  };
7
8  double get_distance(MaterialPoint mp1,
9                      MaterialPoint mp2)
10 {
11   double dx = mp2.x - mp1.x, ...;
12   return std::sqrt( dx * dx + ... );
13 }
14
15 MaterialPoint one = {4, -6, 8},
16                two = {-2, 3, -8};
17 cout << "Расстояние между точками"
18      << get_distance(one, two) << "\n";
```

Использование структур: поле-указатель на саму себя

```
1 struct DayTime
2 {
3   short hour, minute, second;
4
5   // Указателей можно добавлять сколько ↩
        угодно
6   DayTime *prev_moment, *next_moment;
7 };
```

Примеры структур: трёхмерный вектор и возращение значения структуры из функции

```
1  struct Vector3D
2  {
3    double x, y, z;
4  };
5
6
7  // Внутри возращаемого значения — три double
8  Vector3D vec_mult(Vector3D v1, Vector3D v2)
9  {
10   Vector3D v_res = { v1.y * v2.z - v1.z * v2.y,
11                      v1.x * v2.z - v1.z * v2.x,
12                      v1.x * v2.y - v1.y * v2.x };
13   return v_res;
14 }
```

Придумываем задачку: есть сосуд с неким газом, есть образец, который вам захотелось исследовать. Помещаем образец в сосуд и включаем программу, которая контролирует давление газа, стравливая лишнее по необходимости, и температуру, охлаждая/подогревая сосуд.
Внутри идёт какая-то реакция, вы наблюдаете за этим. Для написания управляющей программы, кроме всего прочего, нужно помнить об уравнении состоянии:

$$PV \sim T$$

которое указывает на связь основных параметров.
В данном случае, объём - величина постоянная (сосуд не расширяется), а давление и температура зависят от реакции образца внутри и от окружающей среды (подогреваем или охлаждаем).

Как можно описать программу с использованием структур?

```
1  struct GasContainer
2  { // полей может быть гораздо больше
3    double pressure, temperature;
4    unsigned volume;
5  };
6
7  GasConteiner gs1 = {25, 290.5, 8};
8
9  // Заполняем газом
10 gs1.pressure = 15;
11 gs1.temperature = 299.8;
12
13 // Следим за давлением
14 if ( gs1.pressure > CRIT_PRESSURE ) {
15   release_gas_from( gs1 );
16 }
```

Как можно создать проблему кодом?

```
1  struct GasContainer
2  { // полей может быть гораздо больше
3    double pressure, temperature;
4    unsigned volume;
5  };
6
7  GasConteiner gs1 = {25, 290.5, 8};
8
9  // Заполняем веществом
10 gs1.pressure = 15;
11 gs1.temperature = 292.8;
12
13 gs1.preasure = CRIT_PRESSURE - 10; // тут!
14 if ( gs1.pressure > CRIT_PRESSURE ) {
15   release_gas_from( gs1 );
16 }
```

Последствия

Утечка/взрыв/порча оборудования в отдельно взятой лабаратории

Выводы

- Бывают случаи, когда данные надо защищать от произвольного доступа
- Изменение значения поля составного объекта может требовать более сложной логики (а иногда - и гораздо), чем операция присвоения

Так, в компьютерных наук пришли к тому, что неплохо бы в программах со сложной логикой иметь **объекты** - более продвинутые переменные, чем мы знали прежде. Более того, ведущими умами была разработана концепция написания программ - **объектно-ориентированное программирование** (ООП).
Во многих языках тип данных для подобных объектов назвали **классом**, в том числе и в C++.

Неформальное определение класса

Составной тип данных, аналогичный структурам (**в смысле языка C**), в котором:
а) полями могут быть как другие типы данных, так и функции (получившие термин **метод**);
б) доступ к полям может быть ограничен для кода, использующего переменные (объекты) класса.

Общий вид объявления класса

```
class <название_класса>
{
private:
   <тип_n> <закрытое_поле_n_1>
        [, <закрытое_поле_n_2>, ...];
   <тип> <закрытый_метод>(<аргументы>)
   { <код_функции> };

public:
   <тип_m> <открытое_поле_m_1>
        [, <открытое_поле_m_2>, ...];
   <тип> <открытый_метод>(<аргументы>)
   { <код_функции> };
} [переменная1, переменная2, ...];
```

Использование классов: объявление - по умолчанию все поля закрыты

```
1  class GasContainer
2  {
3    double _pressure, _temperature;
4    double _crit_pressure;
5    size_t _volume;
6  };
7
8  // Так ок, но бесполезно
9  GasContainer gs1;
10
11 //Две строки ниже запрещены
12 //gs1._pressure = 10.5;
13 //gs1._volume = 12;
```

Использование классов: объявление - полный аналог предыдущего слайда

```
1  class GasContainer
2  {
3  private: // необязательная метка
4    double _pressure, _temperature;
5    double _crit_pressure;
6    size_t _volume;
7  };
8
9  // Так ок, но бесполезно
10 GasContainer gs1;
11
12 //Две строки ниже запрещены
13 //gs1._pressure = 10.5;
14 //gs1._volume = 12;
```

Использование классов: объявление - добавляем метод и задаём значения полей в конкретном объекте.

```
1  class GasContainer
2  {
3    double _pressure, _temperature;
4    double _crit_pressure; size_t _volume;
5
6  public:
7    void init(size_t volume, double temp,
8              double crit_pressure)
9    {
10     _temperature = temp; _volume = volume;
11     _pressure = _temperature / _volume;
12     _crit_pressure = crit_pressure;
13   }
14 };
15
16 GasContainer gs1;
17 gs1.init(7, 295.5, 52.5);
```

Использование классов: терминология

- переменную класса - называем **объект**
- функцию в качестве поля - называем **методом класса**
- поля других типов - называем **полями-данными**
- совокупность полей-данных конкретного объекта - образует его **внутреннее состояние**

Использование классов: ещё методы - даём возможность посмотреть значения закрытых полей

```
1  class GasContainer
2  { // описание полей — слайды выше
3  public:
4    void init(size_t volume, double temp,
5              double crit_pressure)
6    { ... } // тело — предыдущие слайды
7
8    double pressure()
9    { return _pressure; }
10 };
11
12 GasContainer gs1;
13 gs1.init(7, 285.5, 50.5);
14
15 cout << "Текущее давление: " << gs1.pressure()
16      << "\n";
```

Использование классов: объект - пост-установка значений

```
1  class GasContainer
2  { // описание полей — слайды выше
3  public:
4    // определения init и pressure выше
5    void set_temp(double new_temp)
6    {
7      if (new_temp < 0) { return; }
8
9      double new_press = new_temp / volume;
10     if (new_press > crit_pressure) {
11       send_alert_msg(new_temp);
12       release_gas();
13     }
14     // Всё ок, меняем температуру
15     _temperature = new_temp;
16     _pressure = _temperature / _volume;
17   }
18 };
```

Использование классов: объект - пост-установка значений

```
1  class GasContainer
2  { // описание полей — слайды выше
3  public:
4    void init(size_t volume, double temp,
5              double crit_pressure)
6    { ... }
7
8    double get_pressure() { ... }
9
10   void set_temp(double new_temp) { ... }
11 };
12
13 GasContainer gs_cur;
14 gs_cur.init(7, 285.5, 50.5);
15 gs_cur.set_temp(312.8);
```

Использование классов: сравнение со структурой

```
1  struct GasContainerOld
2  { ... };
3
4  class GasContainer
5  {
6    // описание полей — слайды выше
7    ...
8  };
9
10 GasContainerOld gs_old = {4, 265.3};
11
12 GasContainer gs1;
13 gs1.init(7, 235.5, 50.5);
14 gs1.init(8, 345.5, 58.5); // Что за???
```

Концепция **конструктора для класса**: специальный метод, который позволяет устанавливать состояние объекта в момент его создания (то есть, при добавлении переменной конкретного класса появляется возможность передать значения закрытым полям).

В C++ **конструкторами** являются методы класса, имя которых **совпадает с названием класса** и которые **не возвращают никакого значения**.

Для любого объекта конструктор может быть вызван только **единожды**.

Использование классов: добавление конструктора

```
1  class GasContainer
2  {
3    double _pressure, _temperature;
4    double _crit_pressure; size_t _volume;
5  public:
6    GasContainer(size_t volume, double temp,
7                 double crit_press) :
8            _temperature{temp}, _volume{vol},
9            _crit_pressure{crit_press}
10   {
11     _pressure = _temperature / _volume;
12   }
13
14   // ...
15 };
16
17 GasContainer gs1{7, 235.7, 85.8};
18 // GasContainer gs2; -> не компилируется
```

Использование классов: почему создание второго объекта **gs2** не компилируется?

- В каждый класс, в котором не объявлено ни одного конструктора неявно добавляется **конструктор по умолчанию**
- **Конструктор по умолчанию** позволяет объявлять переменные класса, не передавая им никаких начальных значений
- **Конструктор по умолчанию** - не принимает никаких аргументов и единственное, что он делает - выделяет нужную память под все поля класса
- Если определён вручную хотя бы один конструктор, неявный конструктор по умолчанию не добавляется в класс
- Для его возращения нужно определить перегруженную функцию конструктора без аргументов

Использование классов: добавление конструктора по умолчанию (перегрузка конструкторов)

```
1  class GasContainer
2  {
3    double _pressure, _temperature;
4    double _crit_pressure; size_t _volume;
5  public:
6    GasContainer(size_t volume, double temp,
7                 double crit_press) :
8            _temperature{temp}, _volume{vol},
9            _crit_pressure{crit_press}
10   {  _pressure = _temperature / _volume; }
11
12   GasContainer() {}
13   // остальные методы
14 };
15
16 GasContainer gs1{7, 385.7, 44.3};
17 GasContainer gs2; // Всё ок технически
```

Страшная правда о структурах в C++

```
1  class GasContainer
2  {
3    double _pressure, _temperature;
4    double _crit_pressure; size_t _volume;
5  };
6
7  // класс выше — тоже самое, что и:
8
9  struct GasContainer
10 {
11 private:
12   double _pressure, _temperature;
13   double _crit_pressure; size_t _volume;
14 };
```

Составные типы данных. Общие советы

- Используйте **структуры** исключительно в смысле языка C - как группировку значений других типов данных в открытых полях. Допустимо добавление простых методов
- Если есть хоть одна причина добавить конструктор для типа - используйте **классы**
- При программировании на C++ любые (свои или библиотечные) структуры или классы в качестве аргументов функций предпочитайте передавать **по ссылке** (возможно, константной) - избегайте ненужного копирования
- При написании классов не определяйте конструктор без параметров, если на то нет веских оснований
- Не экономьте на названиях полей структур/классов (предпочитайте понятные названия кратким сокращениям)

Немного практики: знакомство с **<algorithm>**

Сортировка массивов

Сортировка массива - стандартная задача. В C++ в библиотеке <**algorithm**> определена функция **sort**, с помощью которой можно сортировать массивы различных типов. Её сигнатура следующая:

```
void sort(first, last,
          comparator = operator<);
```

- 1-ый аргумент **first** - указатель (или его аналог) на первый элемент
- 2-ой аргумент **last** - указатель (или его аналог) на элемент, следующий за последним
- 3-ый аргумент **comparator** - функция, которая умеет сравнивать **два** элемента массива. По умолчанию используется оператор «<»

Фактически, сортируется диапазон **[first, last)**.
После работы функции массив, указатели которого были в неё переданы, становится упорядоченным.

Сортировка массивов

Функция сравнения **comparator** должна быть определена как

```
bool comparator(Type elem1, Type elem2);
```

Функция должна возвращать

- **true**, если элемент **elem1** должен идти перед **elem2**
- **false** - иначе

Type - тип сортируемых элементов.

Сортировка массивов

Пример: сортировка действительного массива

```
1  #include <algorithm>
2
3  double my_arr[] = {55.4, 1.34, -0.95, 9.98,
4                     43.56, 3.4};
5
6  sort(my_arr, my_arr + 6);
7
8  cout << "Упорядочение по возрастанию:\n";
9  for (double elem : my_arr) {
10   cout << elem << ' ';
11 }
12 cout << "\n";
```

Сортировка массивов

Пример: сортировка действительного массива по убыванию

```
1  #include <algorithm>
2
3  bool my_compr(double v1, double v2)
4  {
5    return v1 > v2;
6  }
7
8  double my_arr[] = {55.4, 1.34, -0.95, 9.98,
9                     43.56, 3.4};
10
11 sort(my_arr, my_arr + 6, my_compr);
12
13 cout << "Упорядочение по убыванию:\n";
14 for (double elem : my_arr) {
15   cout << elem << ' ';
16 }
17 cout << "\n";
```

Сортировка массивов

Пример: сортировка части массива

```
1  #include <algorithm>
2
3  int ints[] = {3, -4, 11, 67, -2, -1
4                43, 5, 12, -9, 11, -15};
5
6  sort(ints, ints + 7);
7
8  cout << "Упорядочение 6 элементов:\n";
9  for (int elem : ints) {
10   cout << elem << ' ';
11 }
12 cout << "\n";
```

Сортировка массивов

Пример: сортировка стандартного статического массива

```
1  #include <algorithm>
2  #include <array>
3
4  bool ya_compr(double v1, double v2)
5  {
6    return abs(v1) > abs(v2);
7  }
8
9  array<double, 7> reals = {55.4, 1.34, -0.95,
10                           9.98, 43.56, 3.4, -5.67};
11
12 sort(begin(reals), end(reals), ya_compr);
13
14 cout << "Убывание по модулю элементов:\n";
15 for (double elem : reals) {
16   cout << elem << ' ';
17 }
18 cout << "\n";
```

Сортировка массивов

Пример: сортировка стандартного статического массива

```
1  #include <algorithm>
2  #include <vector>
3
4  bool ya_compr(double v1, double v2)
5  { return abs(v1) > abs(v2); }
6
7  vector<double> reals = {55.4, 1.34, -0.95,
8                9.98, 43.56, 3.4, -5.67,
9                -45.6, 18.9, 34.3, -34.2};
10
11 sort(begin(reals), begin(reals) + 5, ya_compr);
12
13 cout << "Частичное убывание по модулю:\n";
14 for (double elem : reals) {
15   cout << elem << ' ';
16 }
17 cout << "\n";
```

Поиск значения в массиве

Библиотека <**algorithm**> определяет функцию **find**, с помощью которой можно искать конкретного значения в массиве. Её сигнатура:

```
type_pointer find(first, last, const type& val);
```

- 1-ый аргумент **first** - указатель (или его аналог) на первый элемент
- 2-ой аргумент **last** - указатель (или его аналог) на элемент, следующий за последним
- 3-ий аргумент **val** - значение для поиска
- возвращаемое значение **type_pointer** - или указатель (аналог) на найденный элемент, или **last**

Поиск идёт в диапазоне **[first, last)**.

Сортировка массивов

Пример: поиск элемента в массиве

```
1  #include <algorithm>
2
3  double my_arr[] = {55.4, 1.34, -0.95,
4                     9.98, 43.56, 3.4};
5  double key;
6
7  cout << "Введите число для поиска: ";
8  cin >> key;
9
10 double *p_found = find(my_arr, my_arr + 6, key);
11
12 if (p_found != my_arr + 6) {
13   cout << key << " найден в массиве\n";
14 } else {
15   cout "массив не содержит " << key << "\n";
16 }
```

Сортировка массивов

Пример: поиск элемента в динамическом массиве

```
1  #include <algorithm>
2  #include <vector>
3
4  vector<double> vec[] = {55.4, 1.34, -0.95,
5                          9.98, 43.56, 3.4};
6  double key;
7
8  cout << "Введите число для поиска: ";
9  cin >> key;
10
11 // Для vector возвращается не указатель!
12 auto p_found = find(begin(vec), end(vec), key);
13
14 if (p_found != end(vec)) {
15   cout << key << " найден в массиве\n";
16 } else {
17   cout "массив не содержит " << key << "\n";
18 }
```

Обмен значениями переменных

<algorithm> предоставляет функцию **swap** для обмена значениями у двух переменных одинакого типа. Её сигнатура:

```
void swap(first_var, second_var);
```

- 1-ый аргумент **first_var** - первая переменная
- 2-ой аргумент **second_var** - вторая переменная

```
1 #include <algorithm>
2
3 // Как обменять значения в лоб
4 double var1 = 10.5, var2 = 45.6;
5 double tmp = var1;
6 var1 = var2;
7 var2 = tmp;
8
9 // а так — с помощью стандартной функции
10 swap(var1, var2);
```

Обмен значениями переменных

Пример: поменять содержимое динамических массивов (**vector**)

```
1 #include <algorithm>
2 #include <vector>
3
4 vector<double> v1 = {2.3, 4.5, 6.7};
5 vector<double> v2 = {1.1, 2.2, 3.3, 4.4, ↩
      5.5};
6
7 swap(v1, v2);
8
9 cout << "Первый массив:\n";
10 for (double elem : v1) {
11   cout << elem << ' ';
12 }
13 cout << "\n";
```