Лекция VI

16 ноября 2018

Пользовательские типы данных

Псевдонимы (aliases)

using (актуальный C++)
typedef (совместимость с прошлыми версиями)

Составные типы

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

Пользовательские типы. Псевдонимы

Актуальный C++ (стандарт C++11 и новее): **using** - добавление псевдонимов

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

Пользовательские типы. Псевдонимы

Актуальный C++ (стандарт C++11 и новее): **using** - добавление псевдонимов

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

Цели использования:

1. Упрощение длинного, сложного или «некрасивого» названия типа.
2. Использование терминологии из предметной области задачи.

using: пример

```
1  using ull_int     = unsigned long long;
2  using ull_int_ptr = unsigned long long*;
3  using tenth_ulls  = unsigned long long[10];
4
5  ull_int     val1 = 5555555555;
6  ull_int_ptr ptr1 = &val1;
7  print(*ptr1);
8
9  tenth_ulls counters = {0};
10 counters[2] = 88;
11 print("Третий счётчик: ", counters[1],'\n');
```

Пользовательские типы. Псевдонимы

using: передача статического массива в функцию по ссылке

```
1  const size_t FOUR = 4;
2
3  using matrix4x4 = double[FOUR][FOUR];
4
5  void fill_matrix_by_rand(matrix4x4& matr)
6  {
7    for (size_t i = 0; i < FOUR; i++) {
8      for (size_t j = 0; j < FOUR; j++) {
9        matr = rand_0_1_incl();
10     }
11   }
12 }
```

Без псевдонимов объявление функции выглядит так:

```
1  void fill_matrix_by_rand(
2          double (&matr)[FOUR][FOUR]);
```

Пользовательские типы. Псевдонимы

Предметно-ориентированные функции

```
1 using amperage_t   = double;
2 using voltage_t    = double;
3 using device_id_t = size_t;
4
5 amperage_t check_current(device_id_t id);
6 voltage_t  check_valtage(device_id_t id);
```

Пользовательские типы. Псевдонимы

Предметно-ориентированные функции

```
1  using angle_t = double;
2
3  double sin_at(angle_t angle)
4  {
5    return sin(angle * M_PI / 180);
6  }
7
8  double cos_at(angle_t angle)
9  {
10   return cos(angle * M_PI / 180);
11 }
12
13 double tg(angle_t angle)
14 {
15   return tan(angle * M_PI / 180);
16 }
17
18 print("tg(45) = ", tg(45), "\n");
```

Пользовательские типы. Псевдонимы

Совместимость с предыдущими версиями стандарта:
typedef - добавление псевдонимов

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

Пользовательские типы. Псевдонимы

C++: оператор **typedef**

```
1 // Объявляем псевдонимы для типа double
2 // и указателя на double
3 typedef double amperage_t, *amperage_ptr;
4
5 amperage_t val1 = 5.55;
6 // Два варианта определения переменных-указателей
7 amperage_ptr p_amp1 = &val1;
```

Полезный совет

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

Переходим к составным типам

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> [, <поле_2>, ...];
  <тип_2> <поле_1> [, <поле_2>, ...];
  ...
  <тип_n> <поле_1> [, <поле_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;
```

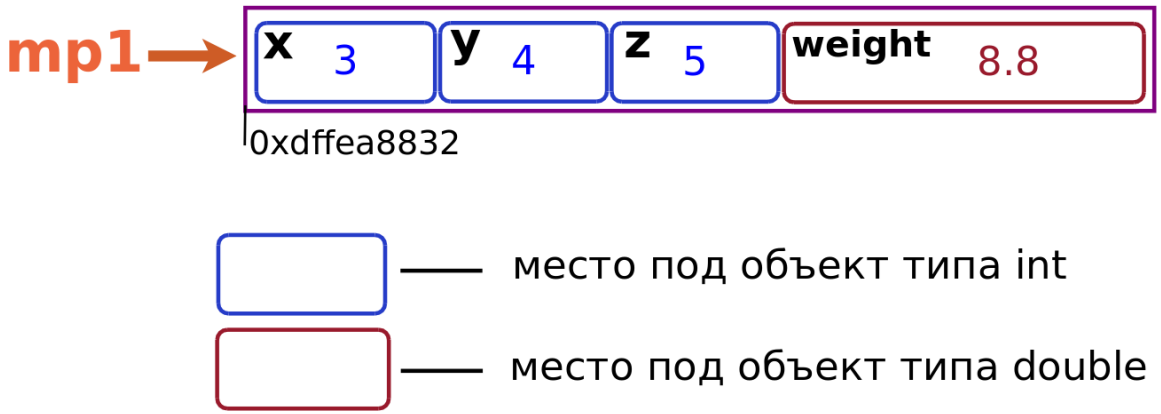
Переменным структуры выделяются блоки памяти, каждый из которых состоит из трёх подблоков размера **int** и одного подблока размера **double**.

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  print("Вес точки: ", mp1.weight, "\n");
10
11 MaterialPoint mp2 = { 5, 8 };
12 // mp2.z == 0, mp2.weigth == 0.0
13 bool is_zero = mp2.z == 0;
14 print("Координата z нулевая? ", is_zero, "\n");
```

При неполной инициализации поля, которым не поставлены никакие значения, получают *значения по умолчанию*.

Схематичное представление переменной структурного типа



Составные типы. Структуры

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

```
1  struct
2  {
3    int state, rank;
4    string name;
5    double factor;
6  } g_var1;
7
8  print("Введите имя: ");
9  getline(cin, g_var1.name);
10
11 g_var1.state = 5;
12 g_var1.rank = -4;
13 g_var1.factor = 5.89;
```

Составные типы. 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 print( (*p_mp).y, "\n" );
11
12 // Получение значения поля по указателю
13 print(p_mp->weight, "\n");
```

Использование структур: параметры функции

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

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 }
```

Примеры структур: отсутствие по умолчанию операторов присвоения и сравнения

```
1  struct Vector3D
2  {
3    double x, y, z;
4  };
5
6  Vector3D first  = {1.0, 1.0, 1.0},
7           second = {2.0, 2.0, 2.0};
8  // Так можно:
9  first = second;
10 // Но нельзя сделать так:
11 // bool is_equal = first == second;
12 // bool is_first_less = first < second;
```

Использование структур: поле-указатель на саму себя; присвоение значений по умолчанию.

```
1  struct ListNode
2  {
3    double vip_data = 0.0;
4
5    ListNode *next = nullptr;
6    ListNode *previous = nullptr;
7  };
8
9  ListNode node1;
10 print("Указатель next равен nullptr? ",
11       node1.next == nullptr, "\n");
```

Резюме по структурам

- Структуры полезны, когда есть данные, которые описывают одну логическую сущность, но сами по себе могут меняться независимо
- Полями могут быть любые известные к моменту объявления сруктуры типы данных
- По умолчанию отсутствуют операторы сравнения
- Полям можно задавать значения по умолчанию

- Не всегда допустимо, чтобы поля составного объекта менялись независимо.
- **Задача** для начала работы с классами: реализовать тип данных, который описывает **квадратную действительную матрицу** размера N.
- Сам тип определяет какие данные необходимы для представления описываемых им объектов, а также какие операции будут допустимы для этих объектов.
- Нужные операции: создание квадратной матрицы, доступ к элементам, расчёт определителя.

Как создаётся квадратная матрица динамического размера?

```
1  double **matrix4 = new double*[4];
2  for (size_t i = 0; i < 4; ++i) {
3    matrix4[i] = new double[4];
4  }
5
6  // ... Работа с матрицей
7
8  for (size_t i = 0; i < 4; ++i) {
9    delete[] matrix4[i];
10 }
11 delete[] matrix4;
```

Как можно описать попробывать использовать структуру?

```
1  struct MatrixNxN
2  {
3    double **matr = nullptr;
4    size_t matrix_size = 0;
5  };
6
7  MatrixNxN matr4x4;
8  // Создание массива
9  matr4x4.matr = new double*[4];
10 for (size_t i = 0; i < 4; ++i) {
11   matr4x4.matr[i] = new double[4];
12 }
13
14 // За что?
15 matr4x4.matrix_size = 6;
```

Выводы

- Бывают случаи, когда данные надо защищать от произвольного доступа
- Изменение значения поля составного объекта может требовать более сложной логики, чем обыкновенная операция присвоения значения

Так, в компьютерных наук пришли к тому, что неплохо бы в программах со сложной логикой иметь **составные типы**, в которых доступ к полям будет ограничен при использовании их переменных в программах.
Во многих языках синтаксическую конструкцию для определения подобных типов назвали **классом**, C++ не исключение.

Неформальное определение класса

Составной тип данных, аналогичный структурам, 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>
{
public:
   <тип_m> <открытое_поле_m_1>
        [, <открытое_поле_m_2>, ...];
   <тип> <открытый_метод>(<аргументы>);

private:
   <тип_n> <закрытое_поле_1>
        [, <закрытое_поле_2>, ...];
   <тип> <закрытый_метод>(<аргументы>);
} [переменная1, переменная2, ...];
```

Создание класса: объявление - по умолчанию все поля закрыты

```
1  class SquareMatrix
2  {
3    double **_matr;
4    size_t _size;
5  };
6
7  // Так ок, но бесполезно
8  SquareMatrix matrix3x3;
9
10 //Две строки приводят к ошибкам компиляции
11 matrix3x3._matr = new double*[3];
12 matrix3x3._size = 12;
```

Создание класса: объявление - полный аналог предыдущего слайда

```
1  class SquareMatrix
2  {
3  private:
4    double **_matr;
5    size_t _size;
6  };
7
8  // Так ок, но бесполезно
9  SquareMatrix matrix3x3;
10
11 //Две строки приводят к ошибкам компиляции
12 matrix3x3._matr = new double*[3];
13 matrix3x3._size = 12;
```

Создание класса: как всё-таки создать матрицу?

```
1  class SquareMatrix
2  {
3  public:
4    void init(size_t N);
5
6  private:
7    double **_matr;
8    size_t _size;
9  };
10
11 void SquareMatrix::init(size_t N)
12 {
13   _matr = new double*[N];
14   for (size_t i = 0; i < N; i++) {
15     _matr[i] = new double[N];
16   }
17   _size = N;
18 }
```

Создание класса: использование метода для инициализации полей

```
1 SquareMatrix matrix3x3;
2
3 matrix3x3.init(3);
```

Создание класса: использование метода для инициализации полей

```
1 SquareMatrix matrix3x3;
2
3 matrix3x3.init(3);

4 matrix3x3.init(7);
```

Создание класса: 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**

Концепция **конструктора для класса**: специальный метод, который позволяет устанавливать состояние объекта в момент его создания (то есть, при создании переменной конкретного класса появляется возможность присваивать 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**.

Создание класса: добавление конструктора

```
1  class SquareMatrix
2  {
3  public:
4    SquareMatrix(size_t N);
5
6  private:
7    double **_matr;
8    size_t _size;
9  };
10
11 SquareMatrix::SquareMatrix(size_t N)
12 {
13   _size = N;
14   _matr = new double*[_size];
15   for (size_t i = 0; i < _size; i++) {
16     _matr[i] = new double[_size];
17   }
18 }
```

Создание класса: добавление конструктора

```
1  class SquareMatrix
2  {
3  public:
4    SquareMatrix(size_t N);
5
6  private:
7    double **_matr;
8    size_t _size;
9  };
10
11 SquareMatrix matrix3x3{3}, matrix5x5{5};
12 // Не компилируется следующая строка:
13 // SquareMatrix rotation;
```

Использование классов: почему создание объекта **rotation** 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
- Если определён вручную хотя бы один конструктор, неявный конструктор по умолчанию не добавляется в класс
- Для его возращения нужно определить перегруженный метод конструктора без аргументов (если логика позволяет)

Создание класса: добавление методов

```
1  class SquareMatrix
2  {
3  public:
4    SquareMatrix(size_t N);
5
6    double get(size_t i, size_t j);
7    void   set(size_t i, size_t j, double value);
8
9    double det();
10   size_t size();
11
12 private:
13   double **_matr;
14   size_t _size;
15
16   double determinant(double **matr, size_t ↩
       matr_size);
17 };
```

Создание класса: добавление методов

```
1  double SquareMatrix::get(size_t i, size_t j)
2  {
3    return _matr[i][j];
4  }
5
6  void SquareMatrix::set(size_t i, size_t j,
7                         double value)
8  {
9    _matr[i][j] = value;
10 }
11
12 size_t SquareMatrix::size()
13 {
14   return _size;
15 }
```

Создание класса: добавление методов

```
1  SquareMatrix matr5x5{5};
2
3  const size_t cur_sz = matr5x5.size();
4  for (size_t i = 0; i < cur_sz; i++) {
5    for (size_t j = 0; j < cur_sz; j++) {
6      matr5x5.set(i, j, rand_a_b_incl(2.5, 7.5));
7    }
8  }
9
10 print("Размер матрицы: ", matr5x5.size(), "\n");
11 print("2-ой элемент 3-ой строки: ",
12       matr5x5.get(2, 1), '\n');
13 print("Определитель: ", matr5x5.det(), "\n");
```

Отлично, создание объектов работает.

Но остаётся вопрос - что происходит в тот момент, когда переменная класса выходит из области видимости?

Возможности классов в C++

В C++ для переменных любых типов можно сформулировать общее правило: **как только переменная выходит** из области видимости, считаем, что выделенная под неё память **освобождена** и может быть переиспользована ОС в других целях.
И тут же вспоминаем - что такое "*выделенная память*"под объект конкретного класса? А это всего лишь память под каждое его поле. C++ каждому составному типу (классы и структуры) предоставляет специальный метод - **деструктор** по умолчанию - который отвечает за автоматическое освобождение памяти при выходе переменных этого типа из области видимости.
В примере с объектами **SquareMatrix** при выходе из области видимости будет удалена память под указатель на **double*** (_matr) и поле типа **size_t** (_size). При этом, вся память выделенная оператором **new** останется занятой до конца работы программы.

Возможности классов в C++

Оставление массивов в памяти, до которых невозможно добраться, является очень плохой практикой при программировании на C++ (и приводит к утечкам ресурсов в ходе работы программы). Но для нашего класса это легко исправить, заменив *деструктор по умолчанию* на свой собственный.
Главные особенности пользовательского деструктора для любого класса в C++:

- деструктор - это специальный метод, который в каждом классе может быть только в одном экземпляре (в отличии от конструкторов);
- имя деструктора строго определено: это знак "тильда"(~) + название класса;
- деструктор как метод не принимает никаких параметров и не возвращает никакого значения.

Создание класса: добавление деструктора

```
1  class SquareMatrix
2  {
3  public:
4    SquareMatrix(size_t N);
5    ~SquareMatrix(); // <- новьё
6
7    double get(size_t i, size_t j);
8    void   set(size_t i, size_t j, double value);
9    double det();
10   size_t size();
11
12 private:
13   double **_matr;
14   size_t _size;
15
16   double determinant(double **matr, size_t ↩
         matr_size);
17 };
```

Создание класса: добавление деструктора

```
1  class SquareMatrix
2  {
3  public:
4    SquareMatrix(size_t N);
5    ~SquareMatrix();
6    ...
7  private:
8    ...
9  };
10
11 SquareMatrix::~SquareMatrix()
12 {
13   for (size_t i = 0; i < _size; i++) {
14     delete[] _matr[i];
15   }
16   delete[] _matr;
17 }
```

Страшная правда о структурах в C++

```
1  class SquareMatrix
2  {
3    double **_matr;
4    size_t _size;
5  };
6
7  // класс выше — тоже самое, что и:
8
9  struct SquareMatrix
10 {
11 private:
12   double **_matr;
13   size_t _size;
14 };
```

Составные типы. 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)

Полная версия класса **SquareMatrix** будет доступна тут:

`https://github.com/posgen/OmsuMaterials/tree/master/2course/Programming/examples/2018_2019`

Немного практики: знакомство с <algorithm>

Обмен значениями переменных

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

`void swap(<тип> first, <тип> second);`

- 1-ый аргумент **first** - первая переменная
- 2-ой аргумент **second** - 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);
```

Выбор наибольшего/наименьшего

<algorithm> предоставляет функции **max** и **min** для выбора, соответственно, максимального и минимального **из двух значений**. Сигнатура:

```
<тип> max(<тип> first, <тип> second);
<тип> min(<тип> first, <тип> second);
```

```
1  #include <algorithm>
2
3
4  double var1 = 10.5, var2 = 45.6;
5  double max_val = max(val, var2),
6         min_val = min(var1, var2);
7
8
9  print("Из чисел ", var1, " и ", var2,
10       " максимальное: ", max_val,
11       "; минимальное: ", min_val, "\n");
```

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  print("Упорядочение по возрастанию:\n");
9  for (double elem : my_arr) {
10  print(elem, ' ');
11 }
12 print("\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 print("Упорядочение по убыванию:\n");
14 for (double elem : my_arr) {
15  print(elem, ' ');
16 }
17 print("\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  print("Упорядочение 6 элементов:\n");
9  for (int elem : ints) {
10   print(elem, ' ');
11 }
12 print("\n");
```

Сортировка массивов

Пример: сортировка динамиеского массива по модулю элементов

```
1  #include <algorithm>
2  const size_t SZ = 12;
3
4  bool ya_compr(double v1, double v2)
5  {  return abs(v1) > abs(v2); }
6
7  double *reals = new double[SZ];
8  for (size_t i = 0; i < SZ; i++) {
9    reals[i] = rand_a_b_incl(-55.5, 55.5);
10 }
11
12 sort(reals, reals + SZ, ya_compr);
13 printf("Убывание по модулю элементов:\n");
14 for (size_t i = 0; i < SZ; i++) {
15   print(reals[i], ' ');
16 }
17 print("\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)**.

Поиск значения в массиве

Пример

```
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  print("Введите число для поиска: ");
8  get_value(key);
9
10 double *last = my_arr + 6;
11 double *p_found = find(my_arr, last, key);
12
13 if (p_found != last) {
14   print(key, " найден в массиве\n");
15 } else {
16   print("массив не содержит ", key, "\n");
17 }
```