Лекция II

22 октября 2016 г.

3. Функции. Начало

Функция - обособленный набор инструкций, пригодный пригодный для повторного использования.
Определение функции в C++

```
[указания_компилятору] <тип_данных>
                       <имя_функции>
                       ( <список_аргументов> )
{
  [инструкции;]
  return <значение>;
}
```

, где **список_аргументов** - это набор пар *тип данных* и *символьное обозначение* каждого параметра, перечисляемых через запятую.

$$n! = 1 * 2 * 3 * 4 ... * n$$

```
1  long long factorial(unsigned n)
2  {
3    long long result = 1;
4
5    for (unsigned i = 2; i <= n; ++i) {
6      result *= i;
7    }
8
9    return result;
10 }
11
12 ...
13
14 long long result = factorial(8);
15 std::cout << "Факториал 8 равен: " << result↩
```

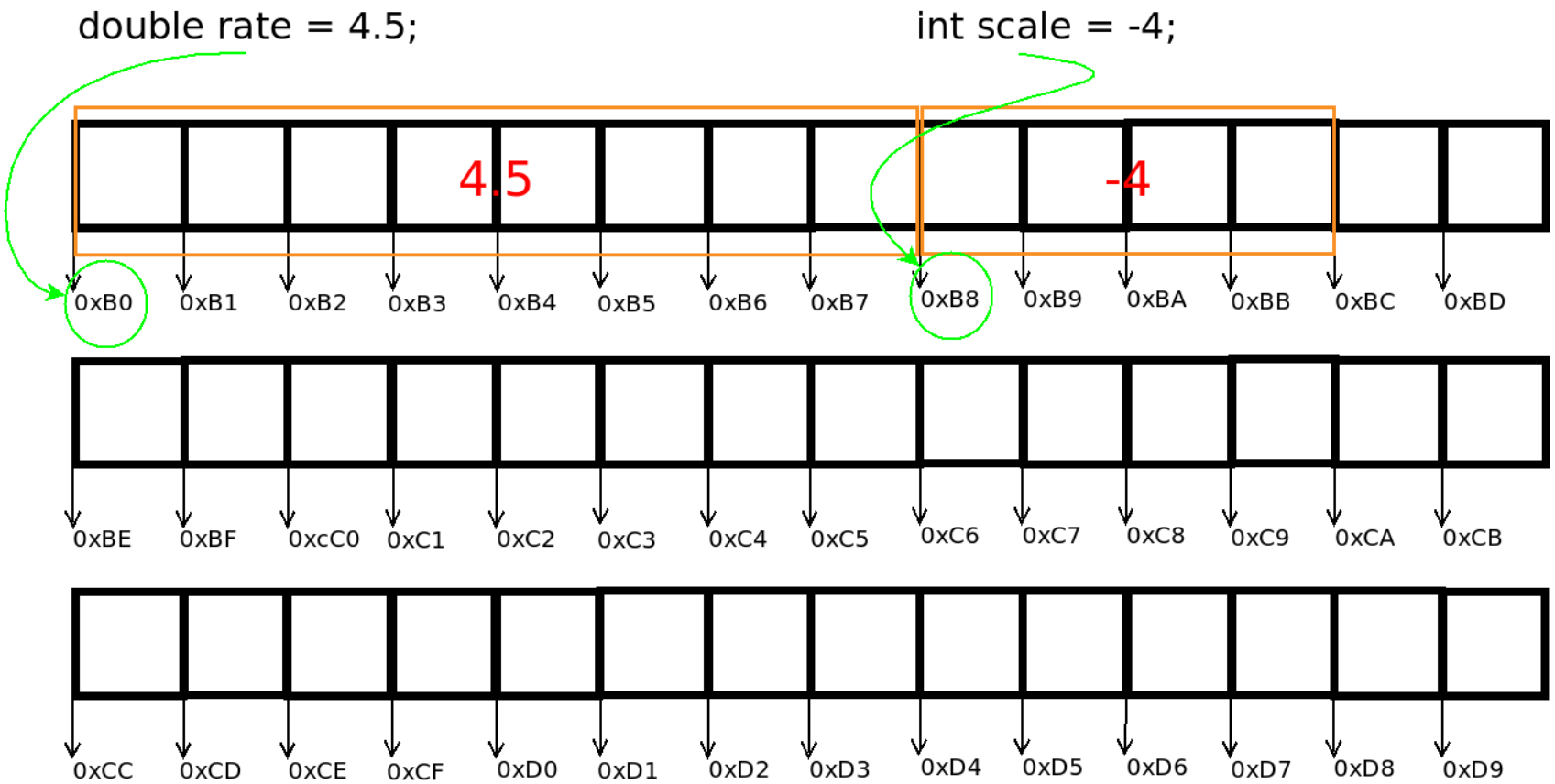
```
1  double degree_nth(double num, int n)
2  {
3    double result = 1;
4    bool is_negative = n < 0;
5    unsigned degree = is_negative ? -n : n;
6
7    while ( degree > 0 ) {
8      result *= num;
9      --degree;
10   }
11
12   return is_negative ? (1 / result) : result↩
         ;
13 }
```

3. Функции. Передача аргументов по значению

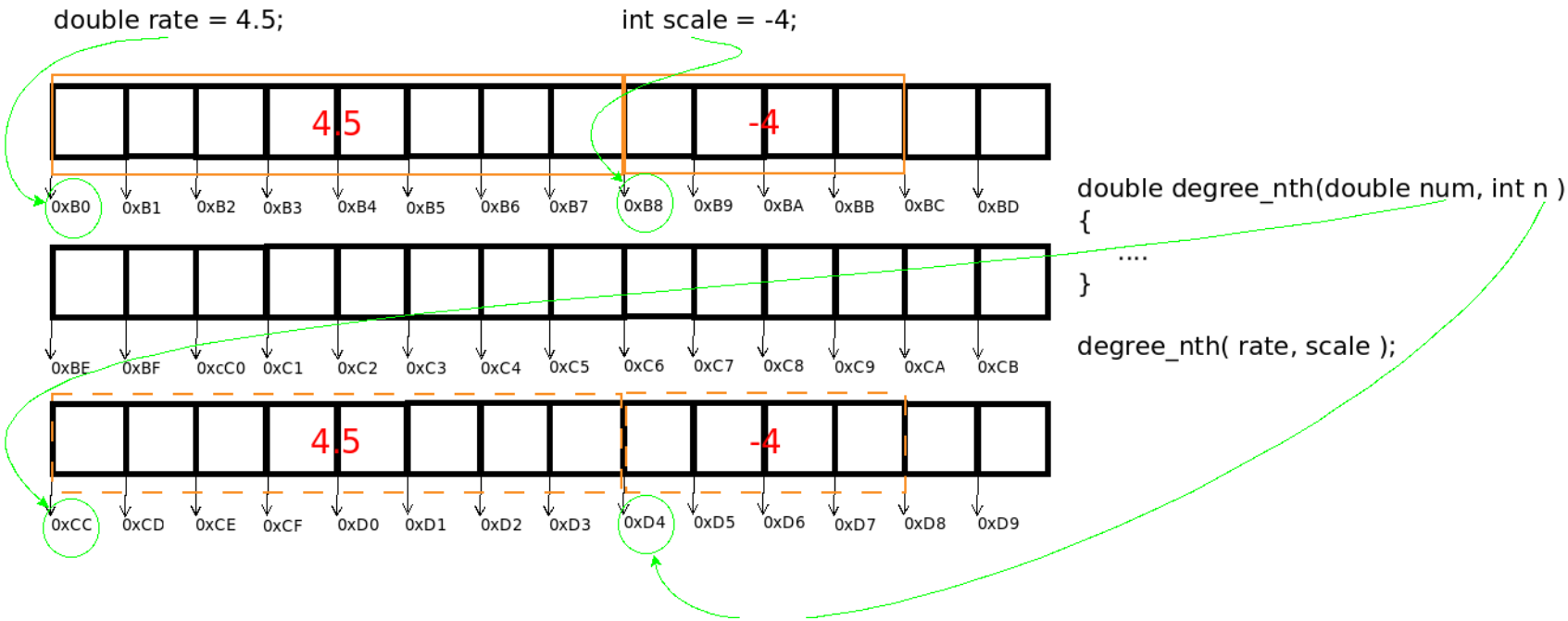
```
1 double degree_nth(double num, int n) { ... }
2
3 double rate = 4.5;
4 int scale = -4;
5
6 double rate_in_scale = degree_nth( rate, ↩
    scale );
```

```
1 double degree_nth(double num, int n)
2 { ... }
3
4 ...
5
6 double rate = 4.5;
7 int scale = -4;
8
9 double rate_in_scale = degree_nth( rate, ↩
    scale );
```

3. Функции. Передача аргументов по значению

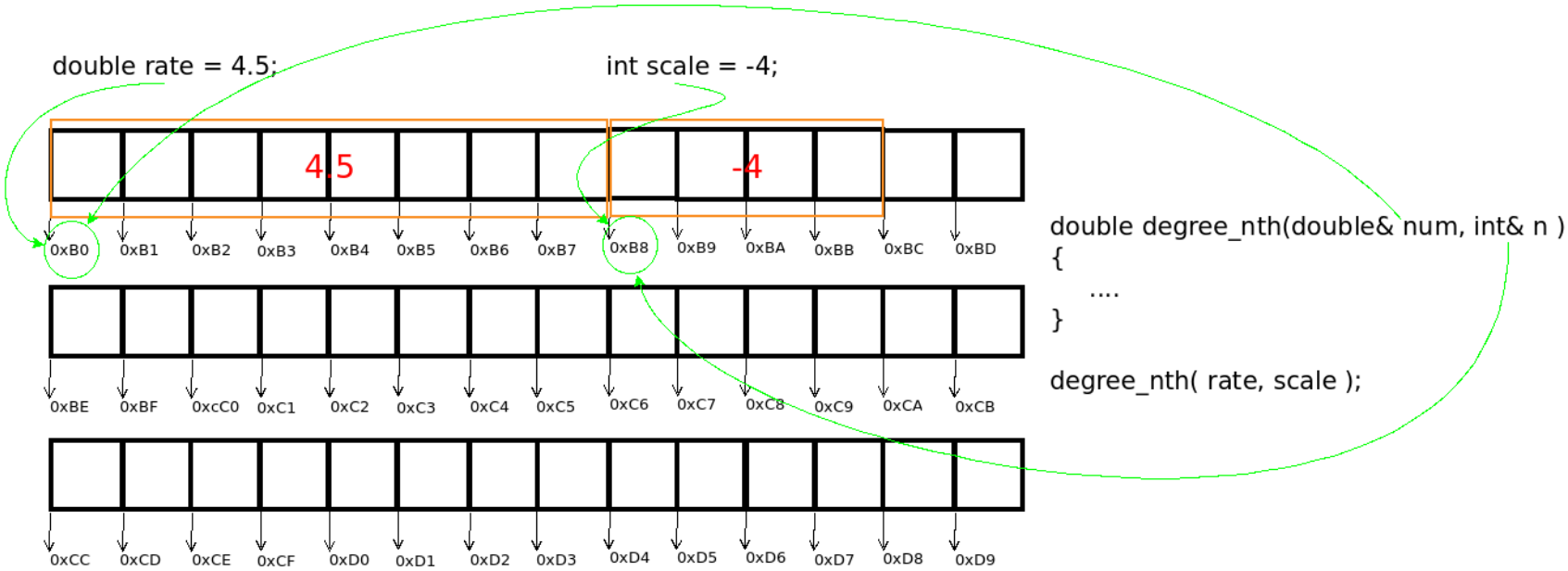


3. Функции. Передача аргументов по значению



```
1 double degree_nth(double& num, int& n)
2 { ... }
3 ...
4
5 double rate = 4.5;
6 int scale = -4;
7
8 double rate_in_scale = degree_nth( rate, ↩
    scale );
```

3. Функции. Передача аргументов по ссылке



Внимание! Если функция принимает аргументы только по ссылке, невозможно в неё передать временные значения. То есть, невозможен вызов вида:

```
9 degree_nth( rate, 5 );
```

```
1  double degree_nth(const double& num,
2                    const int& n)
3  { ... }
4
5  ...
6
7  double rate = 4.5;
8  int scale = -4;
9
10 double rate_in_scale = degree_nth( rate, ↩
       scale );
```

3. Функции. Аргументы по умолчанию

```
1  double degree_nth(double num, int n = 3)
2  { ... }
3
4  ...
5
6  double rate = 4.5;
7
8  // Внутри функции n равно 2
9  std::cout << degree_nth( rate, 2 ) << "\n";
10
11 // Внутри функции n равно 3
12 std::cout << degree_nth( rate ) << "\n";
```

3. Функции. Аргументы по умолчанию

1. Аргументы со значением по умолчанию **должны** идти в конце списка.

```
1 double some_fun1(double r1, double r2, ↩
      double r3, int n1 = 2, int n2 = 3)
2 { ... }
```

2. Такие аргументы не могут быть **ссылочными**.

```
1 // Ошибка компиляции
2 double some_fun2(int& n1 = 2)
3 { ... }
```

```
1  long long factorial(unsigned n)
2  {
3    if ( n < 2) {
4      return 1;
5    }
6
7    long long result = 1;
8
9    for (unsigned i = 2; i <= n; ++i) {
10     result *= i;
11   }
12
13   return result;
14 }
```

3. Функции. Не хочу ничего возвращать

Возможно определять функции не требующие явного вызова **return**. Для такого случая предусмотрен специальный тип данных - **void**

```
1  void print_number(int n)
2  {
3    std::cout << "\nПолучено целое число: " <<↩
         n << std::endl;
4  }
5
6  void print_even_number(int num)
7  {
8    if ( (num % 2) != 0 ) {
9      return;
10   }
11
12   print_number(num);
13 }
```

- Каждая функция должна быть уникальна
- Уникальность функции в языках программирования определяется её **сигнатурой** - набором отличительных признаков
- В C++ сигнатура функции определяется:
  1. Именем
  2. Количеством аргументов (тех. термин *арность*)
  3. Типами аргументов и их порядком

```
1  void print_number(int num)
2  {
3    std::cout << "\nПолучено целое число: " <<↩
         n << std::endl;
4  }
5
6  void print_number(double num)
7  {
8    std::cout << "\nПолучено действительное ↩
       число: " << n << std::endl;
9  }
10
11 ...
12
13 print_number( 56 );
14 print_number( 8.888 );
```

```
1 // математические функции из
2 // стандартной библиотеки
3 #include <cmath>
4
5 ...
6
7 std::abs( 56 );
8 std::abs( -8.888 );
```

Материальная точка: три координаты да масса.

А что если попробывать запрограммировать?

```
1 double get_distance(int x1, int y1, int z1,
2                     int x2, int y2, int z2)
3 {
4   double dx2 = std::pow( x2 - x1, 2),
5          dy2 = std::pow( y2 - y1, 2),
6          dz2 = std::pow( z2 - z1, 2);
7
8   return std::sqrt( dx2 + dy2 + dz2 );
9 }
```

Структура - состовной пользовательский тип данных, состоящий из элементов других типов данных. Каждый элемент называется **полем структуры**.

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double mass;
5  };
6
7  ...
8
9  MaterialPoint p1 = {1, 4, 5, 4.55}, p2;
10 std::cout << "Масса точки: " << p1.mass;
11
12 p2.x = p2.y = p2.z = 5;
13 int another_x = p1.x * 3 - p2.x;
```

```
1  double get_distance(int x1, int y1, int z1,
2                      int x2, int y2, int z2)
3  {
4    ...
5    return std::sqrt( dx2 + dy2 + dz2 );
6  }
7
8  double get_distance(MaterialPoint p1, ↩
       MaterialPoint p2)
9  {
10   double dx2 = std::pow( p2.x - p1.x, 2),
11          dy2 = std::pow( p2.y - p1.y, 2),
12          dz2 = std::pow( p2.z - p1.z, 2);
13
14   return std::sqrt( dx2 + dy2 + dz2 );
15 }
```

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double mass;
5
6    double radius_vector()
7    { return std::sqrt(x*x + y*y + z*z); }
8  };
9
10 ...
11
12 MaterialPoint p1 = {6, 4, 5, 8.55};
13 std::cout << "Масса точки: " << p1.mass;
14 std::cout << "\nРадиус-вектор: " << p1.↩
     radius_vector();
```

Массивы в C++. Наследие C

Массив - структура данных, содержащая набор проиндексированных элементов. В языке C++ массивы являются типизированными, то есть могут содержать элементы только одного типа.

```
<тип_данных> <имя_массива>[<размер>];
```

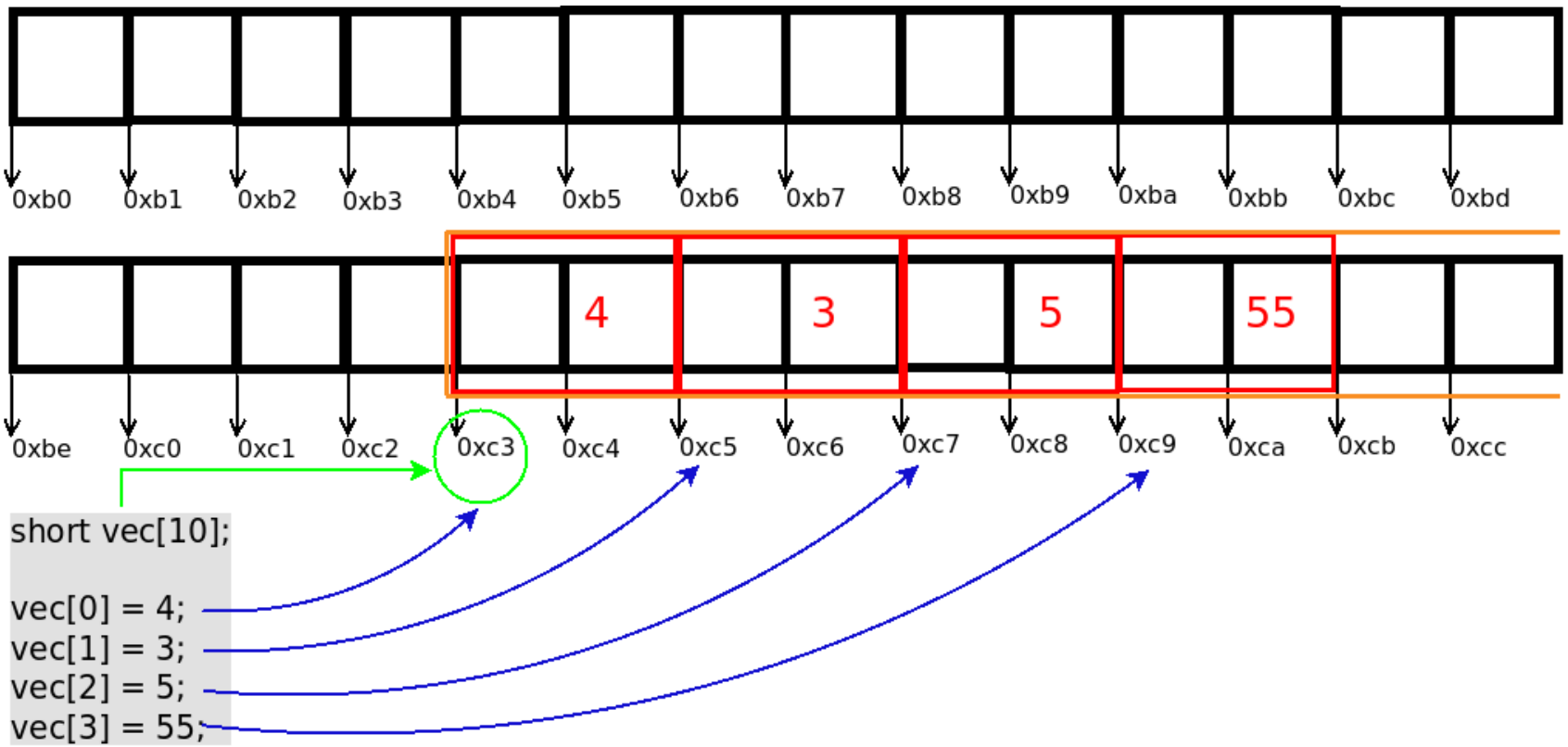
В C++ все элементы одного массива располагаются в памяти последовательно.

Статический массив

Индексация элементов в массиве **начинается с нуля**.

```
1 short vec[10];
2
3 // Задаём значение первого элемента
4 vec[0] = 4;
5 vec[3] = 55;
6
7 std::cout << vec[0];
```

Точка зрения памяти



Инициализация массива

```
1  int vec[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, ↩
       10 };
2
3  for (unsigned i = 0; i < 10; ++i) {
4    std::cout << vec[i] << " ";
5  }
6
7  // Незаданные элементы будут равны 0
8  double real_arr[5] = { 3.4, 5.5, 77.11 };
9
10 // При явной инициализации размер массива ↩
       можно пропускать
11 int another_vec[] = { 1, 2, 3, 4 };
```

Многомерные массивы

```
1  int matrix1[10][10];
2
3  for (int i = 0; i < 10; ++i) {
4    for (int j = 0; j < 10; ++j) {
5      matrix[i][j] = i + j;
6    }
7  }
8
9  // Инициализация
10 int matrix2[3][3] = { {1, 2, 3}, {4, 5, 6}, ↩
     {7, 8, 9} };
```

Массивы в C++. Наследие C

Многомерные массивы: в числе размерностей никто не ограничен. Гарантируется работа до 31 уровня вложенности.

```
1 int monstr[3][4][5][3][4][5];
2 monstr[0][0][0][0][0][0] = 5;
```

Бонусы C++11: специальный цикл **for** для прохода по массиву (for-range).

```
1  double rates[] = { 1.1, 2.2, 3.3, 5.0, 6.555↩
       };
2
3  // Вывод элементов массива на экран
4  for (double r : rates) {
5    std::cout << r << " ";
6  }
7
8  // Изменение значения каждого элемента ↩
       массива
9  for (double& r : rates) {
10   r *= r;
11 }
```

Бонусы C++11: специальный цикл **for** для прохода по массиву (for-range).

```
1 double rates[] = { 1.1, 2.2, 3.3, 5.0, 6.555↩
      };
2
3 for (double& r : rates) {
4   r *= r;
5 }
```

Ограничения

Данный вариант цикла **for** работает только тогда, когда объявление массива и цикл находятся в **одной** области видимости.

Передача массивов в функции.

```
1  void print_array(int arr[], size_t count)
2  {
3    std::cout << "\nПереданный массив:\n";
4    for (unsigned i = 0; i < count; ++i) {
5      std::cout << arr[i] << " ";
6    }
7    std::cout << "\n";
8  }
9
10 ...
11
12 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
13 print_array(vec, 8);
```

Никакой проверки размерности массива не происходит.

```
1  void print_array(int arr[55], size_t count)
2  {
3    std::cout << "\nПереданный массив:\n";
4    for (unsigned i = 0; i < count; ++i) {
5      std::cout << arr[i] << " ";
6    }
7    std::cout << "\n";
8  }
9
10 ...
11
12 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
13 print_array(vec, 8);
```

Массивы в C++. Наследие C

Многомерные массивы: нужно указать все размерности, кроме первой.

```
1  const size_t COLS = 8;
2
3  void print_2D_array(int matr[][COLS], size_t rows_count)
4  {
5    std::cout << "\nПереданный массив:\n";
6    for (unsigned i = 0; i < rows_count; ++i) {
7      for (unsigned j = 0; j < COLS; ++j) {
8        std::cout << matr[i][j] << " ";
9      }
10     std::cout << "\n";
11   }
12   std::cout << "\n";
13 }
```

Массивы в C++. Наши дни

Начиная с C++11 реализация статического массива фиксированной размерности доступна в стандартной библиотеке **<array>**

```
1 #include <array>
2 ...
3
4 std::array<int,10> points;
5 points[2] = 4;
6
7 std:array<double,3> rates = {0.1, 0.2, 0.3};
8 std::cout << "Размер rates: ";
9 std::cout << rates.size();
```

Массивы в C++. Наши дни

```
1 #include <array>
2 ...
3
4 std:array<double,3> rates = {0.1, 0.2, 0.3};
5 std::cout << "\nПервый элемент: " << rates.↩
     front();
6 std::cout << "\nПоследний элемент: " << ↩
     rates.back();
```

Массивы в C++. Наши дни

```
1  #include <array>
2  ...
3  const size_t SZ = 3;
4
5  void print_array(std:array<double, SZ> arr)
6  {
7    for (double elem : arr) {
8      std::cout << elem << " ";
9    }
10 }
11
12
13 std:array<double, SZ> rates = {0.1, 0.2, ↩
       0.3};
14 print_array( rates );
```

Массивы в C++. Наши дни

```
1 #include <array>
2 const size_t SZ = 3;
3 ...
4
5 // 2D массив
6 std:array<std:array<double, SZ>, SZ> matrix;
7 matrix[1][1] = 55;
```