Лекция II

29 сентября 2017

3. Функции. Определение

Функция - обособленный набор инструкций, пригодный для повторного использования, который может принимать *произвольное количество* значений и возвращать *одно* значение.
Определение функции в C++

```
[...] <тип_возвращаемого_значения>
      <имя_функции> ( <список_аргументов> )
{
  [инструкции;]
  return <значение>;
}
```

, где **список_аргументов** - это набор пар *тип данных* и *символьное обозначение* каждого параметра, перечисляемых через запятую. Троеточия - специальные указания компилятору, возможно рассмотрятся на практике.

3. Функции. Определение

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

```
1  unsigned long long factorial(unsigned n)
2  {
3    unsigned 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 unsigned long long result = factorial(8);
14 cout << "Факториал 8 равен: " << result;
```

3. Функции. Множественный return

Число операторов возврата **return** внутри функции - неограничено.

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

number^n, n — целое

```
1  #include <cstdlib>
2
3  double degree_nth(double num, int n)
4  {
5    double result = 1;
6    // Берём модуль от числа n
7    unsigned degree = abs(n);
8
9    while ( degree > 0 ) {
10     result *= num;
11     --degree;
12   }
13
14   return (n < 0) ? (1 / result) : result;
15 }
```

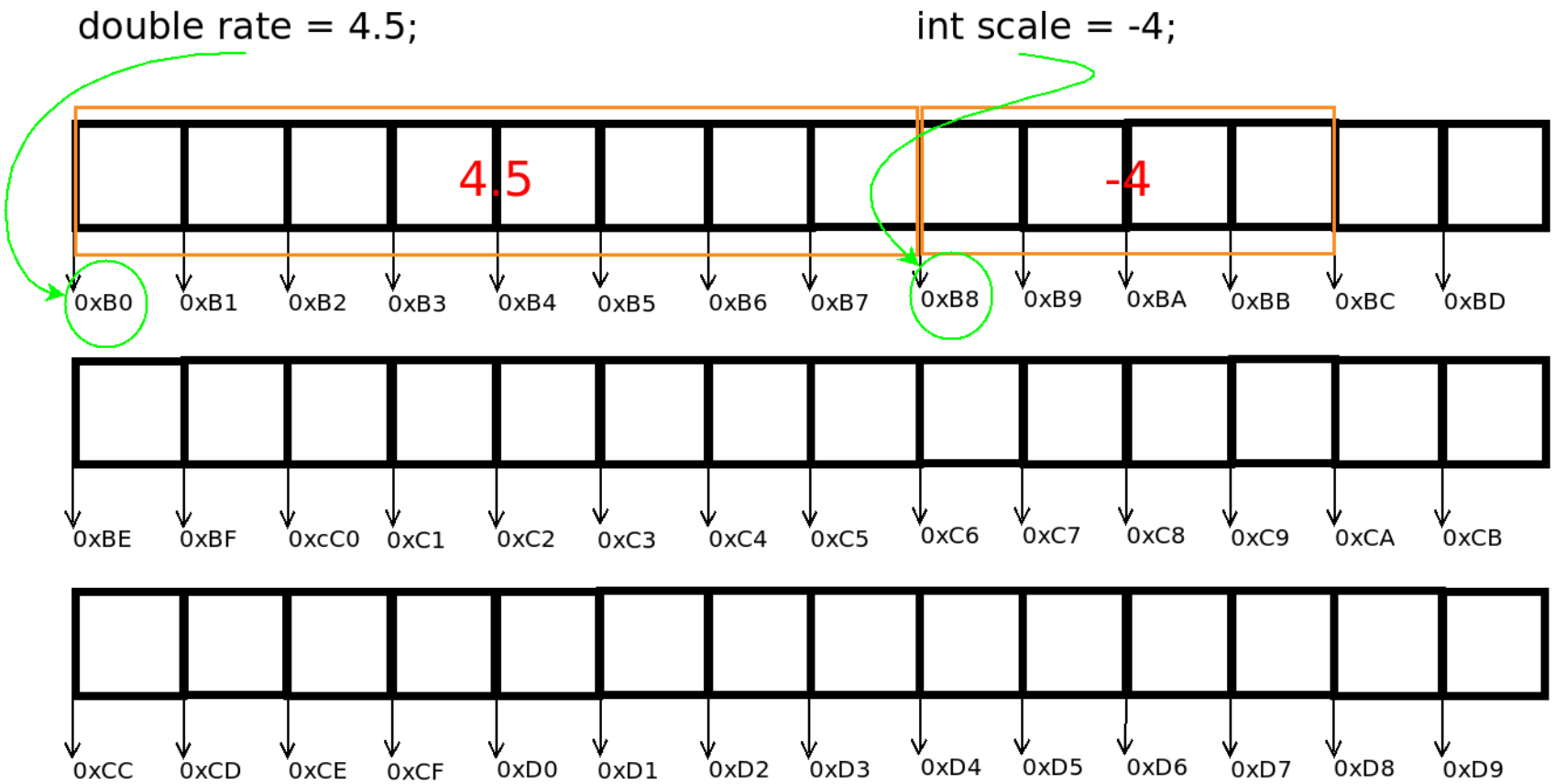
```
1 double degree_nth(double num, int n)
2 { ... }
3
4 double rate  = 4.5;
5 int    scale = -4;
6
7 double rate_in_sc = degree_nth(rate, scale);
8 cout << degree_nth(8.85, -30) << "\n";
```

Прежде, чем идти дальше

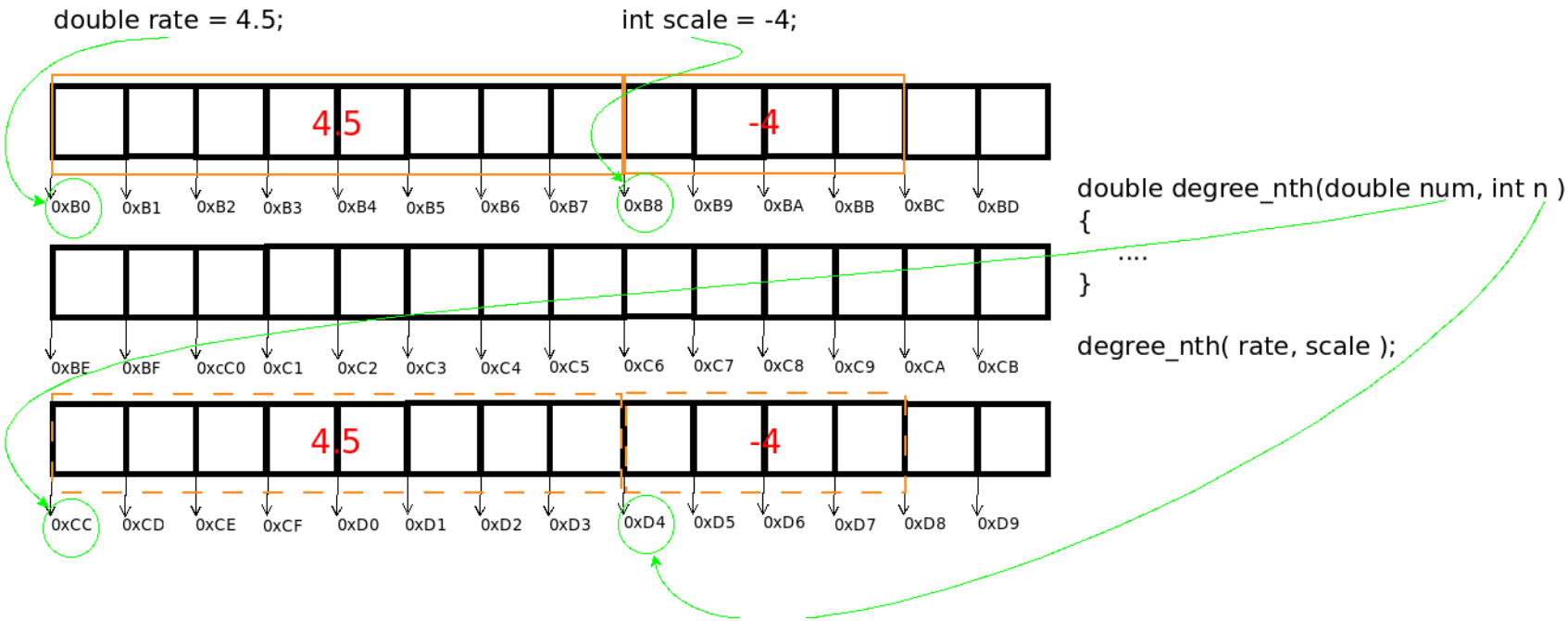
Адрес переменной

Каждая переменная любого типа связана с блоком в памяти, который состоит из какого-либо количества байт. Каждый байт в модели памяти языка C++ пронумерован - ему присвоен порядковый номер: 0, 1, 2, 3, ... Номер первого байта блока называется **адресом переменной**. Адреса обычно приводят в *шестнадцатиричном* виде.

3. Функции. Передача аргументов по значению



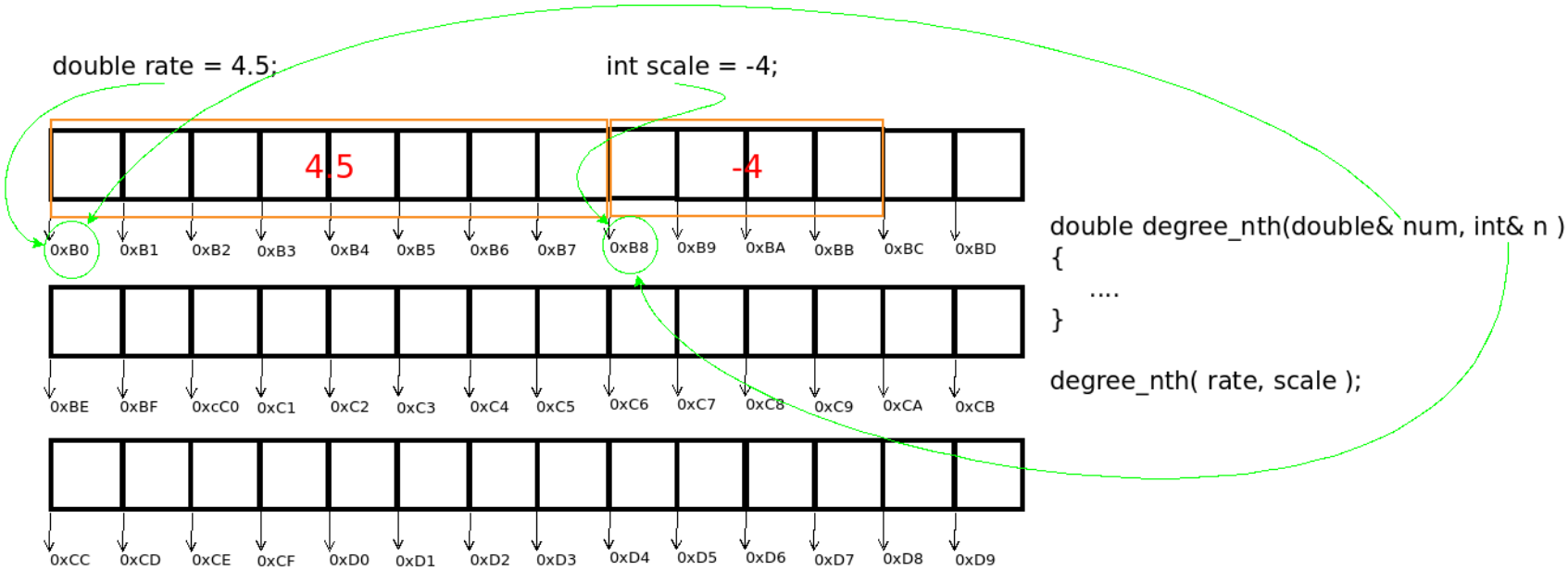
3. Функции. Передача аргументов по значению



3. Функции. Передача аргументов по ссылке

```
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_sc = degree_nth(rate, scale);
```

3. Функции. Передача аргументов по ссылке



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

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

Аргументы, передающиеся как по ссылке, так и по значению, можно сделать неизменяемыми (константными).

```
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 );
```

Пример: запросить у пользователя n целых чисел, найти максимальное и каким по счёту оно было введено

```
1  int find_max_with_pos(unsigned count,
2                        unsigned& place)
3  {
4      int current, max_num;
5
6      for (unsigned i = 1; i <= count; ++i) {
7          cout << "Введите " << i << " число: ";
8          cin >> current;
9
10         if ((i > 1) && (current > max_num)) {
11           current = max_num; place = i;
12         } else {
13           max_num = current; place = 1;
14         }
15     }
16
17     return max_num;
18 }
19 ...
20 unsigned position;
21 int max_elem = find_max_with_pos(5, position);
```

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,
2                  double r3, int n1 = 2,
3                  int n2 = 3)
4 { ... }
```

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

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

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    cout << "\nПолучено число: " << n << "\n";
4  }
5
6  void print_even_number(int num)
7  {
8    if ( (num % 2) != 0 ) {
9      return;
10   }
11
12   print_number(num);
13 }
14 // А так — нельзя:
15 // void var_of_void;
```

- Каждая функция должна быть уникальна
- Уникальность функции в языках программирования определяется её **сигнатурой** - набором отличительных признаков
- В C++ сигнатура функции определяется:
  1. Именем (идентификатором)
  2. Количеством аргументов (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  /*
2    Получение модуля целого или
3    действительного числа
4  */
5  #include <cmath>
6  #include <cstdlib>
7
8  ...
9
10 std::abs( 56 );
11 std::abs( -8.888 );
```

Аналогично переменным, для функций существуют понятия **объявления** и **определения**. Определение - это все примеры выше, а под **объявлением** понимается указание *типа возвращаемого значения, названия функции и перечисление типов всех аргументов*. Причём, давать имена аргументам - не обязательно

```
1  #include <iostream>
2
3  double my_random(unsigned); // Объявление!
4
5  int main()
6  {
7    cout << "Случайное число: " << my_random(567)
8         << "\n";
9  }
10
11 double my_random(unsigned seed) // Определение!
12 { return 1234.5688 / seed; }
```

Резюме по функциям

- Количество аргументов выбираем мы, но вернуть можем только одно конкретное значение
- Два способа передачи аргументов - **по значению** и **по ссылке**
- Специальный тип **void** - когда из функции не нужно возращать никакого значения
- Название функции не уникально - помним о перегрузке

Прежде, чем идти дальше

Область видимости идентификатора

Место в файле с исходным кодом, где идентификатор доступен для операций (использование значения в случае переменной, вызов - в случае функции). В языке C различают две области - **глобальная** и **локальная**.

- Все функции имеют **глобальную** область видимости
- Переменная имеет **локальную** область видимости, если её объявление или определение находится внутри пары фигурных скобок ({})
- Иначе переменная получает **глобальную** область видимости
- Локальная переменная видна во всех вложенных областях

Прежде, чем идти дальше

Область видимости идентификатора

```
1  // Переменная с глобальной областью видимости
2  const double PI = 3.14159265358;
3
4  void do_something(double x)
5  {
6    // Локальная область видимости
7    double rate = 0.5;
8
9    if (x > 10.0) {
10     // Ещё одна. PI и rate тут доступны
11     double num = PI * x * rate;
12     cout << "Результат: " << num << "\n";
13   }
14
15   // Доступа к num уже нет:
16   // num += 3.05;
17   rate *= 9.5;
18 }
```

Прежде, чем идти дальше

Время жизни переменной

- Переменные с локальной областью видимости автоматически удаляются по достижении конца области видимости
- Глобальные переменные существуют до тех пор, пока выполняется программа

Материальная точка: три координаты да масса.
Найдём расстояние между двумя точками.

```
1  #include <cmath>
2
3  double get_distance(int x1, int y1, int z1,
4                      int x2, int y2, int z2)
5  {
6    double dx2 = pow(x2 - x1, 2),
7           dy2 = pow(y2 - y1, 2),
8           dz2 = pow(z2 - z1, 2);
9
10   return sqrt(dx2 + dy2 + dz2);
11 }
```

Структура - составной 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 some_val = 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: "
15           << p1.radius_vector();
```

Для начала про составные типы данных хватит информации

Далее - первый специальный тип данных, **массив**

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

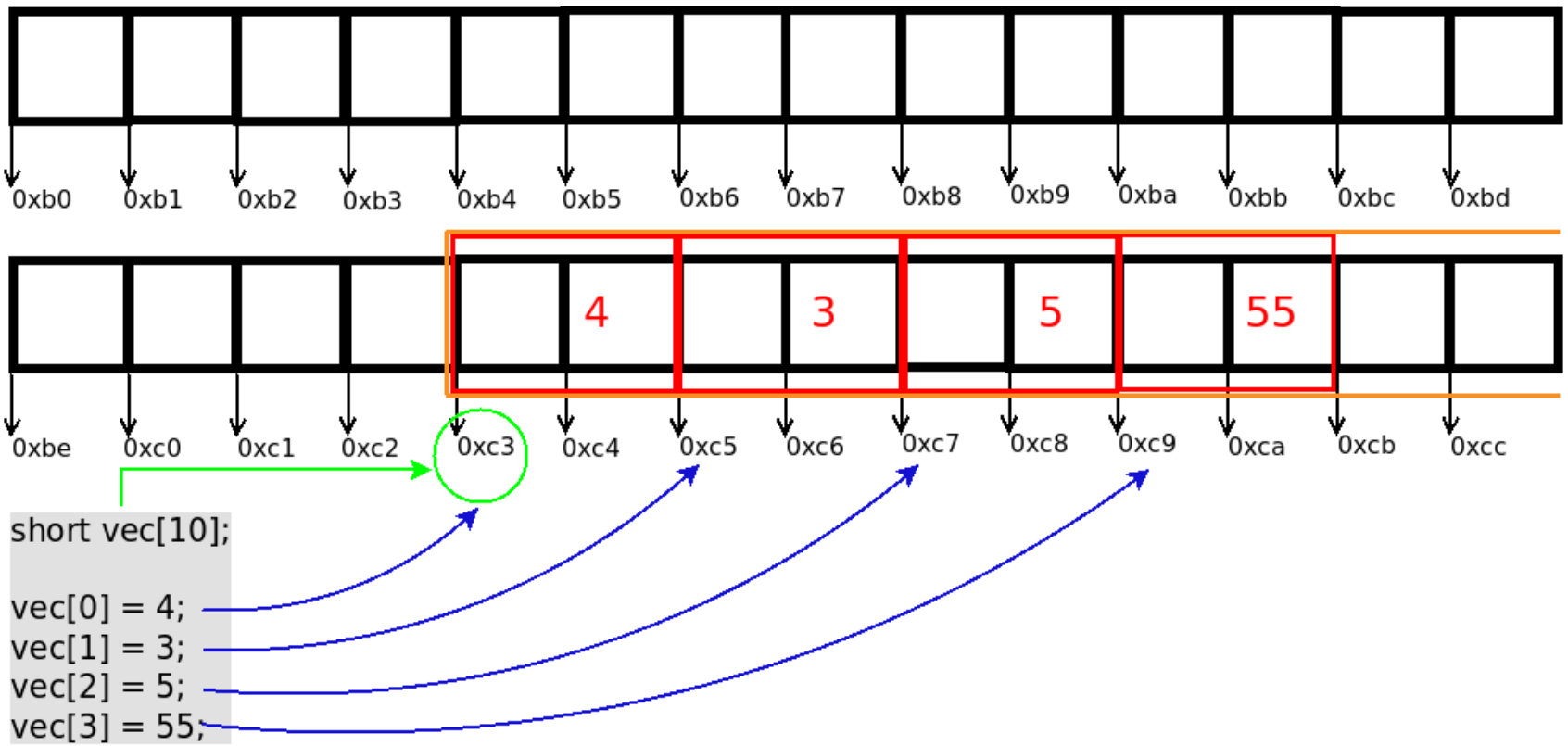
Индексация элементов в массиве **начинается с нуля**.

```
1 short vec[10];
2
3 // Задаём значение первого элемента
4 vec[0] = 4;
5 vec[3] = 55;
6
7 cout << vec[0];
8
9 // cout << vec[15]; // А так лучше не делать
```

Замечание об индексации

Язык C++ не определяет, что должно происходить при применении индекса, который *превышает* размер массива. Такая ситуация называется **неопределённым поведением**. Компиляторы также не имеют возможности проверить индекс.

Точка зрения памяти



Инициализация массива (присвоение начальных значений его элементам)

```
1  int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
2
3  for (unsigned i = 0; i < 8; ++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      matrix1[i][j] = i + j;
6    }
7  }
8
9  // Инициализация
10 int matrix2[3][3] = { {1, 2, 3},
11                       {4, 5, 6},
12                       {7, 8, 9} };
```

Многомерные массивы: печать двумерного массива

```
1  #include <iomanip>
2  // Размер лучше делать глобальной константой
3  const unsigned SZ = 10;
4  ...
5
6  int matrix[SZ][SZ] = { ... };
7
8  cout << "Заданная матрица:\n";
9  for (int i = 0; i < SZ; ++i) {
10   for (int j = 0; j < SZ; ++j) {
11     cout << setw(5) << matrix[i][j] << " ";
12   }
13   cout << "\n";
14 }
15 cout << "\n";
```

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** (for-range). С массивами в стиле C используется для перебора всех элементов цикла.

```
1  double rates[] = { 1.1, 2.2, 5.2, 6.5 };
2
3  // Вывод элементов массива на экран
4  for (double r : rates) {
5    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, 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 в качестве аргумента. Размер одномерного массива можно не указывать

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

Никакой проверки размерности массива не происходит.

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

Кроме того, фактически при передаче массива в функцию происходит не копирование всего массива, а копирование только адреса первого элемента.

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    cout << "\nПереданный массив:\n";
6    for (size_t i = 0; i < rows_count; ++i) {
7      for (size_t j = 0; j < COLS; ++j) {
8        cout << matr[i][j] << " ";
9      }
10     cout << "\n";
11   }
12   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 std::array<int, 10> points;
4 points[2] = 4;
5
6 array<double, 3> rates = {0.1, 0.2, 0.3};
7 cout << "Размер rates: ";
8 cout << rates.size();
```

Массивы в C++. Наши дни

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

Массивы в C++. Наши дни

```
1  const size_t SZ = 3;
2
3  void print_array(array<double, SZ>& arr)
4  {
5    for (double elem : arr) {
6      cout << elem << " ";
7    }
8  }
9
10 array<double, SZ> rates = {0.1, 0.2, 0.3};
11 print_array( rates );
12
13 array<double, SZ + 2> more_rates = {1.1};
14 // Так уже не получится:
15 // print_array( more_rates );
```

Массивы в C++. Наши дни

```
1  #include <array>
2  const size_t SZ = 3;
3  ...
4
5  // 2D массив, состоящий из
6  // 3–х строк и 6 столбцов !!!
7  std:array<std:array<double, SZ * 2>, SZ> matrix;
8  matrix[1][1] = 55;
9
10 // Но индексы, увы, и тут не проверяются
11 cout << matrix[10][10] << "\n";
12 // Не повторять!
```