Лекция V

2 ноября 2018

Адреса, переменные и память
Углублённое расследование

Адрес переменной

Каждая переменная любого типа в C++ связана с сопоставленным ей блоком в оперативной памяти. Длина блока изменяется в байтах, для разных типов - различна.
Адресом переменной называют **номер первого байта** из блока, который отведён под неё. Это целое положительное число.
У переменной можно узнать адрес, в которой она располагается, с помощью оператора - **&**

```
1 int scale = -4;
2 double rate = 2.64;
3
4 print("Адрес rate: ", &rate, "\n");
5 print("Адрес scale: ", &scale, "\n");
```

Возможный вывод:

```
Адрес rate: 0x7fffb2dc22a0
Адрес scale: 0x7fffb2dc2280
```

Адрес переменной

Более того, у каждой переменной или типа можно узнать **длину блока** в байтах с помощью оператора **sizeof**

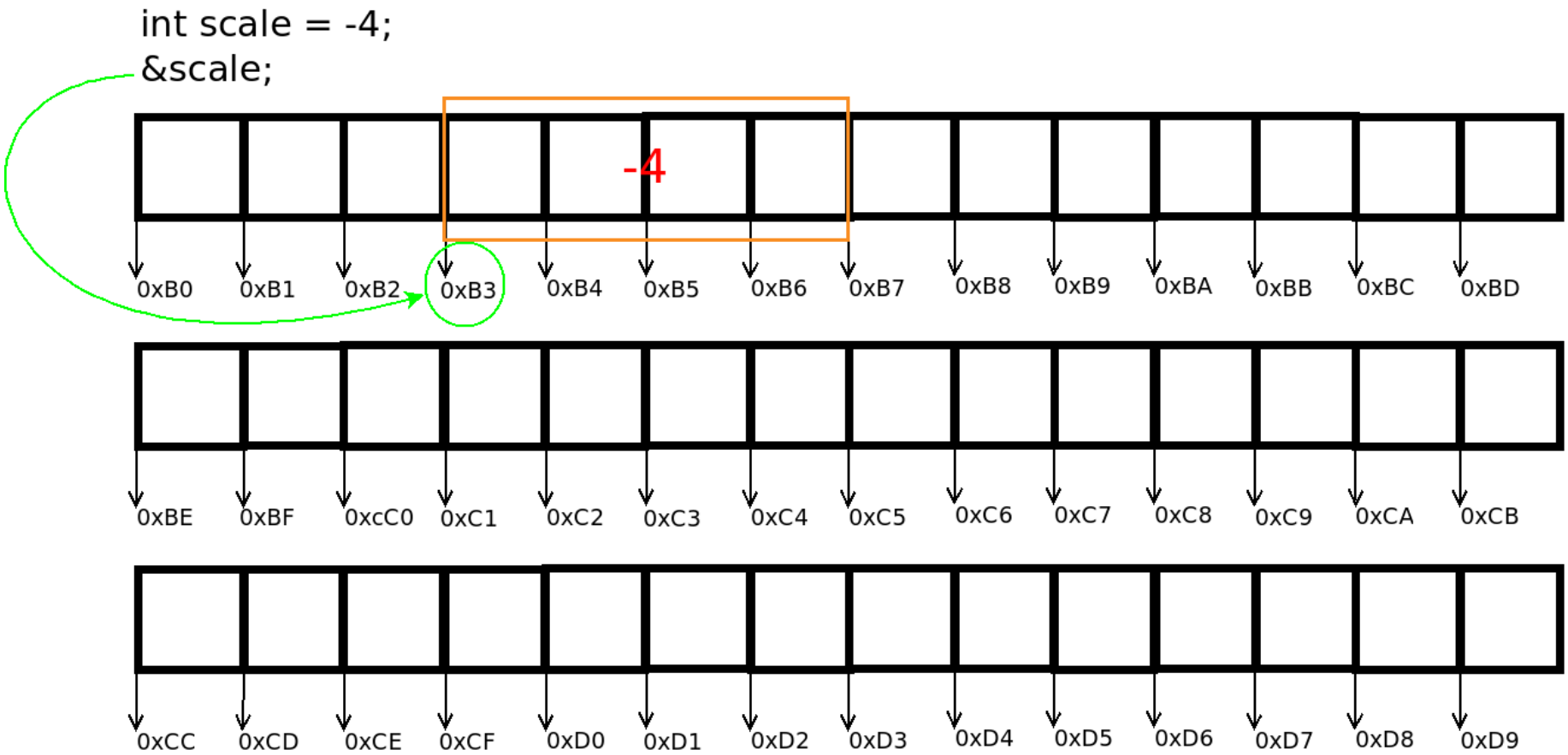
```
1 int scale = -4;
2 double rate = 2.64;
3
4 print("Размер int    : ", sizeof(int), "\n");
5 print("Размер double: ", sizeof(rate), "\n");
6
7 string str = "Тестируем и строку";
8 print("Размер string: ", sizeof(str), "(!?)\n");
```

Возможный вывод:

```
Размер int    : 4
Размер double: 8
Размер string: 32(!?)
```

Адрес переменной

Как адрес выглядит графически: переменная **scale** имеет адрес равный **0xB3** и состоит из четырёх байт.



Указатели

Указателем (pointer) - называют тип данных, переменные которого предназначены для хранения адресов других объектов (*переменных фундаментальных, специальных или пользовательских типов*) и манипуляции с ними[адресами]. Указатели в C++ являются типизированными.

Синтаксис объявления указателя

```
<тип_данных> * <имя_переменной>;
```

Указатели

Указателем (pointer) - называют тип данных, переменные которого предназначены для хранения адресов других объектов (*переменных фундаментальных, специальных или пользовательских типов*) и манипуляции с ними[адресами]. Указатели в C++ являются типизированными.

Синтаксис объявления указателя

```
<тип_данных> * <имя_переменной>;
```

```
1 int    * p_int;    // указатель на int
2 char*    p_char;   // указатель на char
3 double *p_double;  // указатель на double
```

Указатели

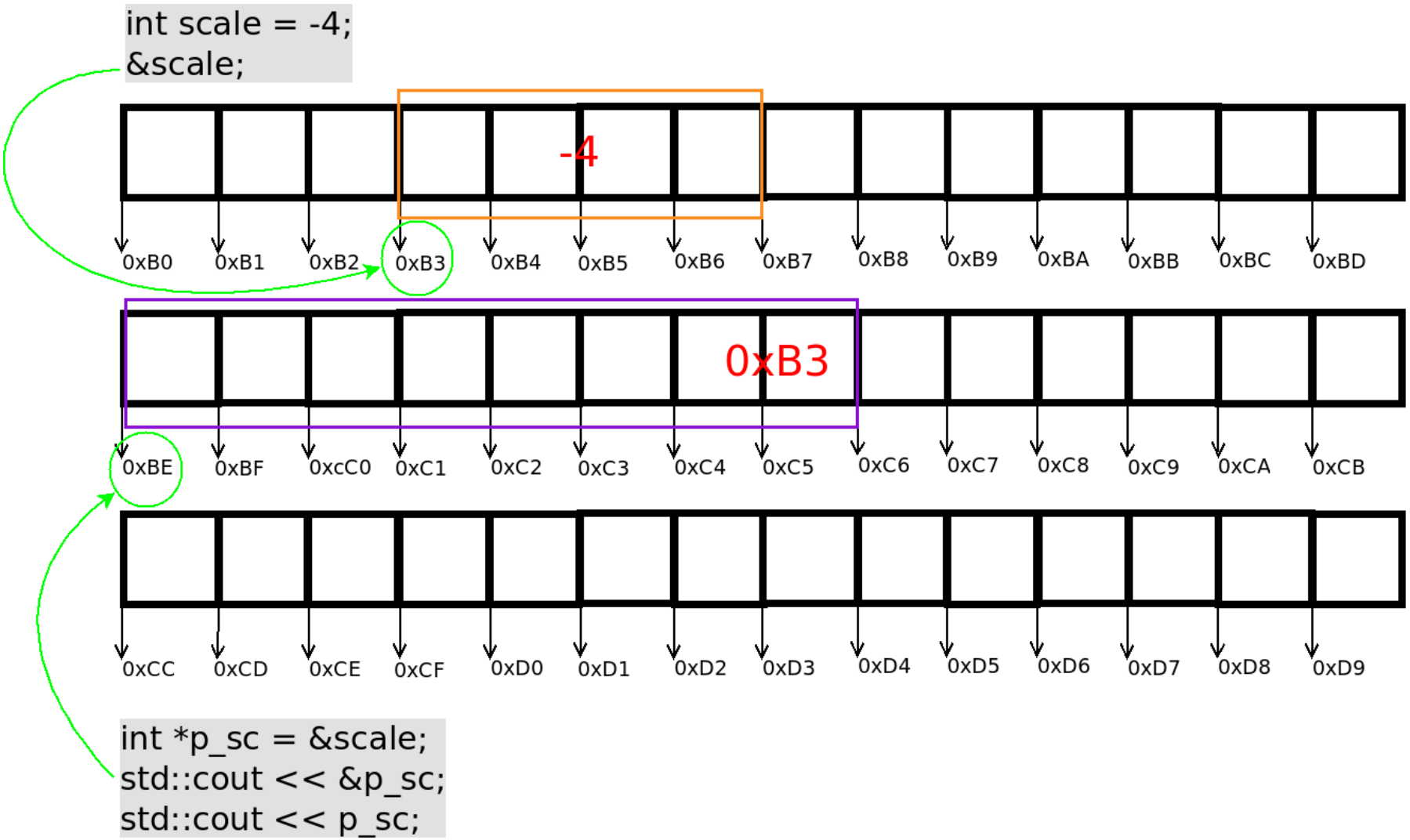
Операции с указателями: **присвоение значения «=»**
Указателю может быть присвоено значение (адрес в памяти) либо с помощью операции взятия адреса у переменной, либо копированием значения из другого указателя.

```
1 int scale = -4, *p_sc;
2
3 p_sc = &scale;
4 int *p2 = p_sc;
```

1-я строка: определяем переменную целого типа **scale** и указатель на целое **p_sc**.
3-я строка: присваиваем указателю **p_sc** значение, равное адресу ячейки памяти, в которой находится переменная **scale**.
4-я строка: присваиваем указателю **p2** значение, которое находится в указателе **p_sc**.

Указатели

В памяти картина следующая. Обратите внимание, сама по себе **p_sc** - просто переменная, со своим адресом.



Указатели

Существует специальное значение для указателей, которое означает, что сама переменная-указатель не содержит реальный адрес какого-либо объекта. Для обозначения такого значения в C++ (начиная со стандарта C++11) применяется ключевое слово **nullptr**, равное некоторой константе. Она известна как **нулевой адрес**. Сам указатель с таким значением называют **нулевым указателем**.

```
1 int *p2 = nullptr;
2
3 // Как-нибудь меняем p2 ...
4
5 if (p2 == nullptr) {
6   print("Указатель p2 так и не получил "
7         "осмысленного значения\n");
8 }
```

Правило для работы с указателями: переменная-указатель **всегда** должна быть *определена*, а не *объявлена*. То есть, ей надо или присвоить реальный адрес, или значение **nullptr**.

Указатели

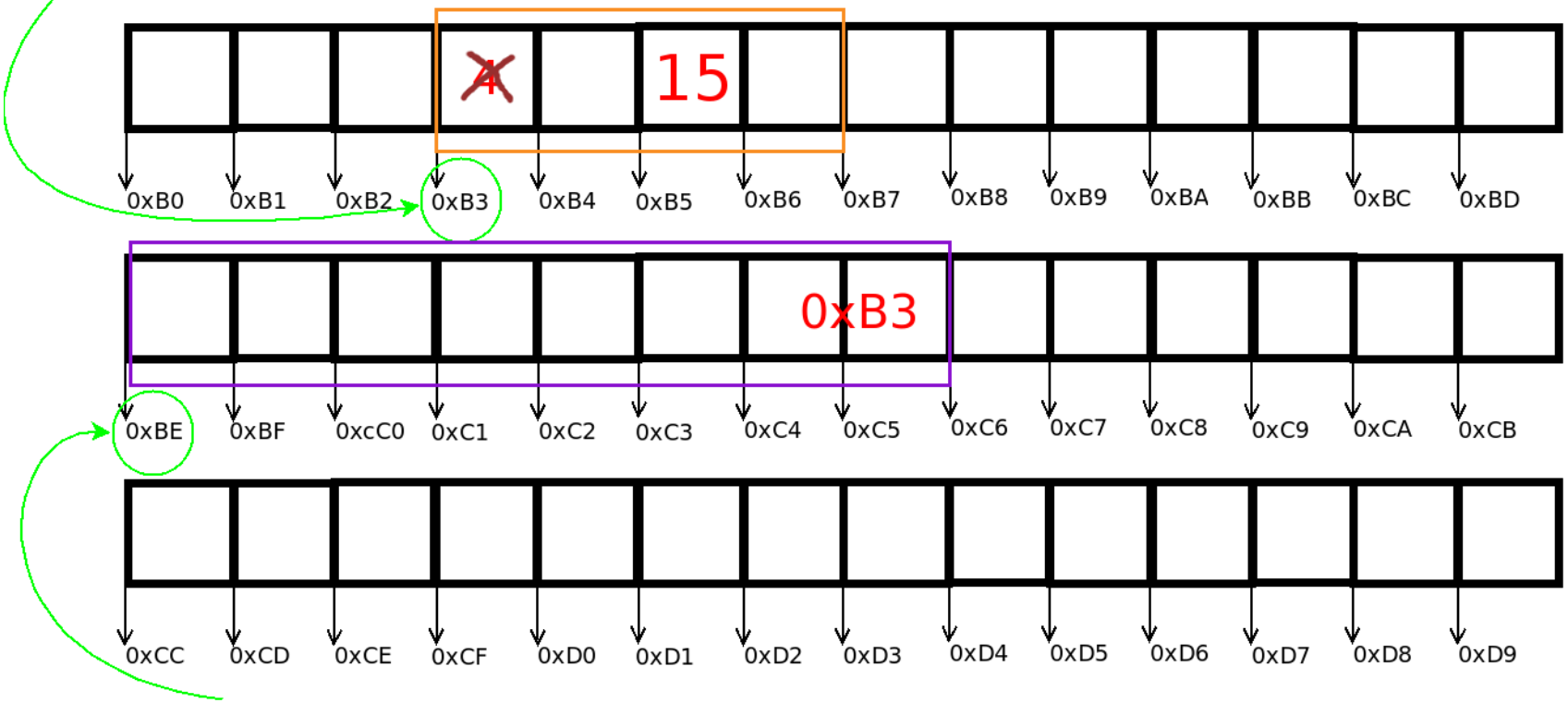
Операции с указателями: **разыменование «*»** - получение самой переменной, адрес которой сохранён в указателе, для операций чтения или изменения её значения.

*<имя_переменной_указателя>;

```
1  int scale = 4;
2  int *p_sc = &scale;
3
4  // Вывод 4 на экран
5  print("Значение, на которое ссылается ",
6        "p_sc: ", *p_sc, "\n");
7  // При разыменовании переменная-указатель
8  // участвует в арифметических выражениях
9  int rate = (*p_sc) + 15;
10
11 // изменяем значение scale
12 *p_sc = 15;
13 // Печатаем 15
14 print("scale = ", scale, "\n");
```

Схематично разыменование можно показать так:

```
int scale = 4;
&scale;
```




```
int *p_sc = &scale;
std::cout << *p_sc;
*p_sc = 15;
```

Операции с указателями: **разыменование**

Предупреждение: разыменование объявленной, но не определённой, переменной-указателя **чрезвычайно опасно** в коде на C++. Тоже самое верно и для указателей, которым присвоен *нулевой указатель* – **nullptr**.

Так никогда не делать!

```
1 double *p_real;
2 print("Что за число тут: ", *p_real, "\n");
3
4 int *p_int = nullptr;
5 print("Может быть повезёт: ", *p_int, "\n");
```

Указатели и функции

Переменные-указатели, как и обычные переменные C++, могут использоваться в качестве параметров функций.

- по умолчанию происходит передача указателя в функцию **по значению**: создаётся отдельная переменная-указатель, в неё копируется адрес передаваемого;

```
1 void make_pointer_happy(int *pointer);
2 int scale = 10;
3 int *p1 = &scale;
4 make_pointer_happy(p1);
```

- но возможна и передача указателя **по ссылке**: для этого указывается «**&**» после символа «*****»;

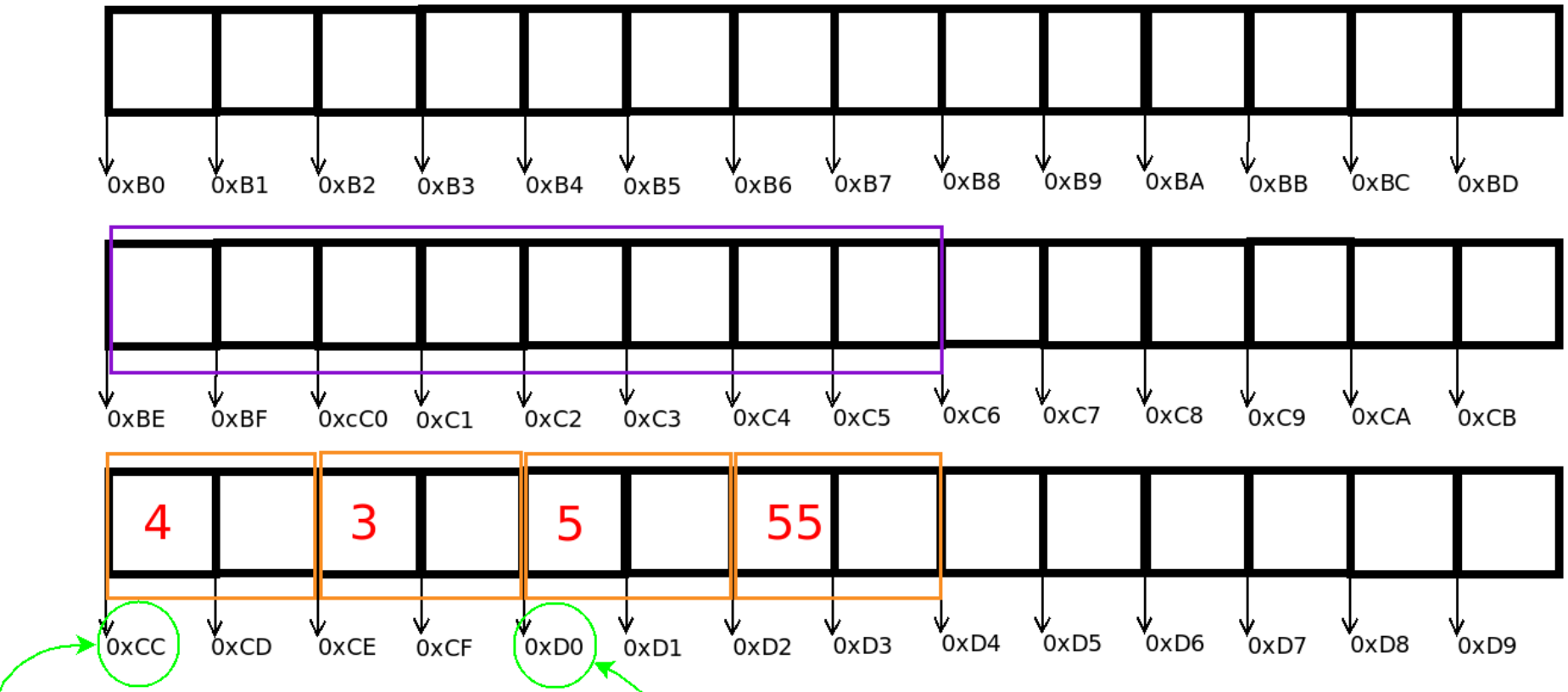
```
1 void pass_pointer_by_ref(int *& pointer);
```

- с помощью оператора взятия адреса «**&**» из обычной переменной можно получить указатель.

```
5 make_pointer_happy( &scale );
```

Указатели и статические массивы

Вспомним, как массив располагается в памяти




```
short vec[4] = {4, 3, 5, 55};

&vec[0];

&vec[2];
```

Указатели и статические массивы

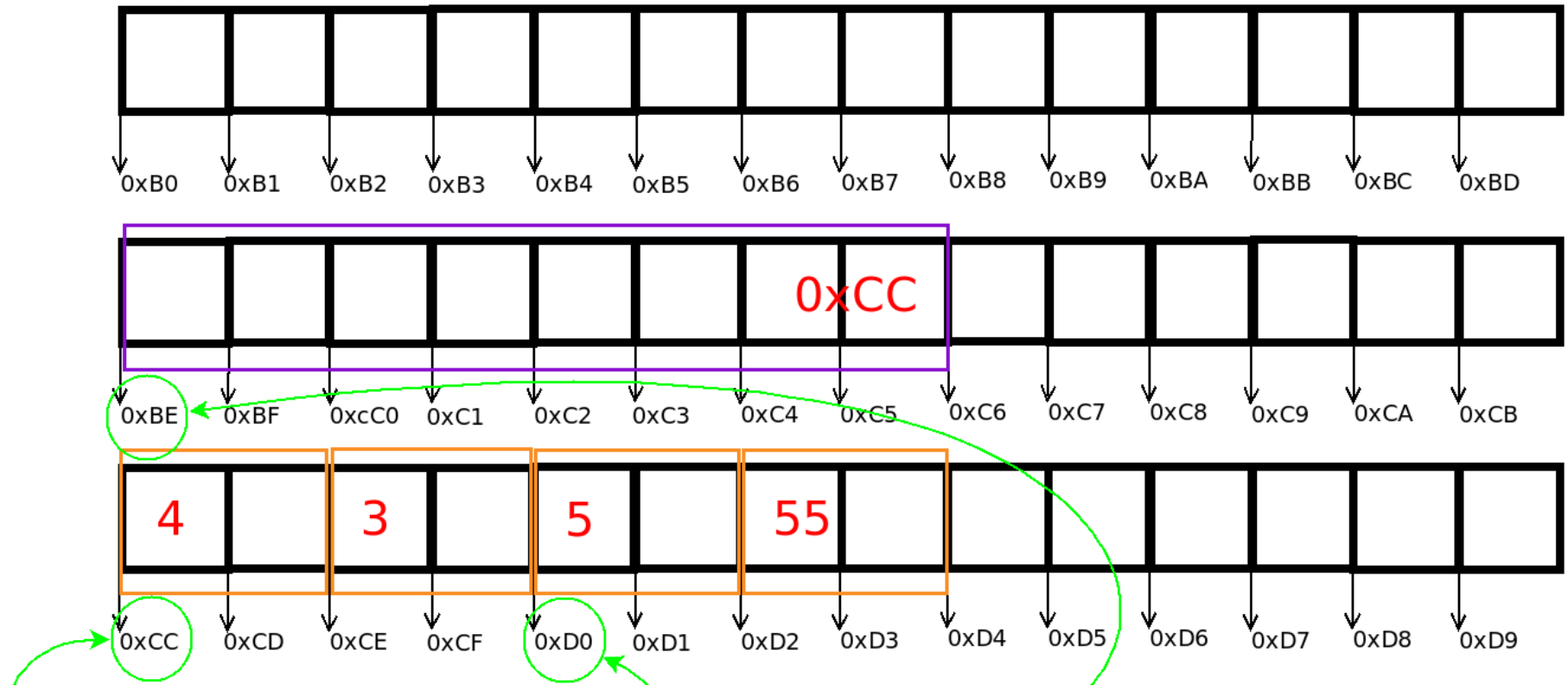
Сходства и различия указателей и массивов

- Имя переменной-массива (выше - **vec**) является указателем на его первый элемент
- Массивы, при передаче в функцию *по значению*, фактически ведут себя как указатели
- Переменной массива **нельзя** присвоить никакой другой адрес (в отличии от переменной-указателя)
- Указатель может быть использован в качестве возвращаемого значения из функции, массив - нет

```
1 // С позапрошлой лекции
2 void print_array(int* arr, size_t count);
3 ...
4 int vec[4] = {4, 3, 5, 55};
5 print_array(vec, 4);
6 // +Можно делать так:
7 int *p_vec = vec;
```

Указатели и статические массивы

Картина в памяти: в указатель записали адрес массива




```
short vec[4] = {4, 3, 5, 55};

&vec[0];

&vec[2];
```

```
short *p_vec = vec;
std::cout << *p_vec;
```

Указатели и статические массивы

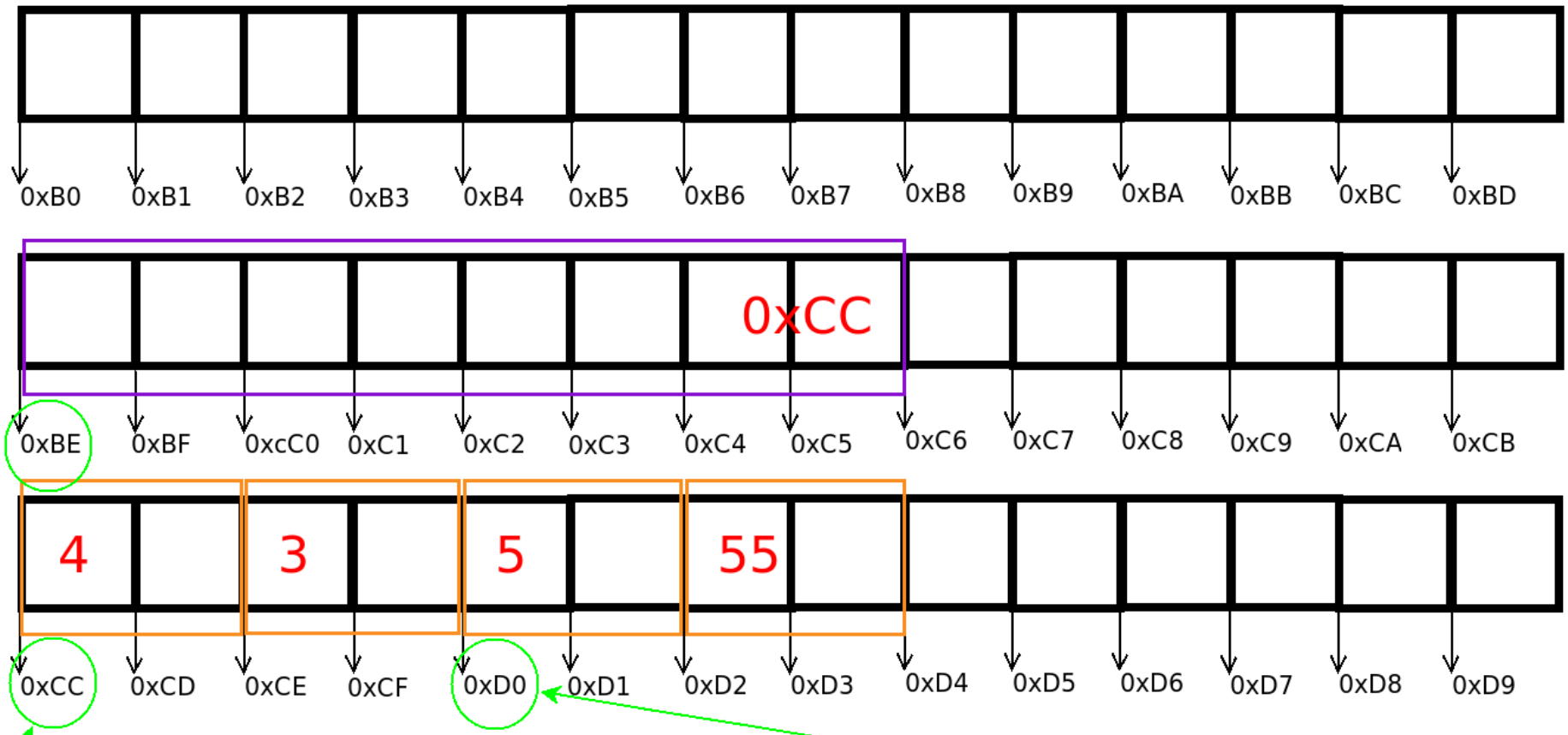
Операции с указателями: **сложение с целым числом**: результатом операции прибавления целого числа **n** к указателю является новый указатель, значение которого смещено на $n * \text{sizeof}(< \text{type} >)$ байт (вправо или влево - зависит от знака **n**).

```
1 short vec[4] = {4, 3, 5, 55};
2 short p_vec = vec;
3
4 // Печатаем 5
5 print("Значение третьего элемента: ",
6       *(p_vec + 2),"\n");
```

Смещение происходит блоками, размер которого определяется типом указателя (указатель на **int**, **double**, **char** и прочие типы).

Указатели и статические массивы

Сложение указателя с целым числом графически




```
short vec[4] = {4, 3, 5, 55};

short *p_vec = vec;
p_vec;
p_vec + 2;
```

Указатели и статические массивы

Также определены операции инкремента / декремента / вычитания целых чисел

```
1  short vec[4] = {4, 3, 5, 55};
2  short *p_vec = vec;
3
4  // Прибавляем единицу — указываем на второй ↩
       элемент
5  p_vec++;
6
7  // теперь — на третий
8  p_vec += 1;
9
10 // и обратно, к первому элементу массива
11 p_vec -= 2;
```

Указатели и статические массивы

Операции с указателями: **индексация**

Индексация указателя выполняет два действия:

1. Сместиться на количество блоков, равных индексу, от текущего адреса
2. Получить доступ к блоку (переменной) по адресу, полученному в результате **смещения**

```
1 int vec[4] = {4, 3, 5, 55};
2 int *p_vec = vec;
3
4 if (p_vec[2] == *(p_vec + 2)) {
5   print("Значения равны\n");
6 }
7
8 print("Четвёртый элемент: ", *(vec + 3), "\n");
```

Указатели и статические массивы

Операции с указателями: **вычитание однотипных указателей**
Результатом вычитания является целое число (как положительное, так и отрицательное), показывающее количество блоков памяти между двумя адресами.

```
1  int vec[4] = {4, 3, 5, 55};
2  int *p1 = vec, *p2 = &vec[3];
3
4  int diff = p2 - p1;
5  // Печатает 3
6  print("Между первым и последним ",
7        "элементом массива расположены ",
8        diff, "элемента\n");
9
10 diff = p1 - p2;
11 // Печатает -3
12 print("Обратно: ", diff, "\n");
```

Указатели и динамическое управление памятью

Указатели и динамическая память

Что входит в понятие **динамическое управление памятью**?

- Язык C++ предоставляет операторы для получения от ОС блоков памяти, заданного размера
- Размер задаётся в **количестве** элементов определённого типа
- Динамический блок памяти существует до тех пор, пока не будет вызван оператор для его возвращения в ОС (либо программа завершит работу)
- Доступ к таким блокам осуществляется только при помощи указателей

Управление динамической памятью

полезно, так как в различных задачах нужны блоки памяти под переменные или массивы, время жизни которых должно превышать локальную область видимости (пример - очереди, стеки, списки, многомерные массивы). В задачах вычислительной физики - особенно.

Указатели и динамическая память

Оператор **new** - запрос **одного** блока динамической памяти у ОС, размер которого равен размеры запрашиваемого типа (C++11). При успешном выполнении, оператор возвращает **указатель на выделенный блок**

```
(1) new <тип>;
(2) new (std::nothrow) <тип>;
```

```
1  int *p1;
2 p1 = new int;
3 // При ошибке выделения: выход из программы
4 *p1 = 89;
5
6 int *p2 = new (nothrow) int;
7 // Явная проверка — доступна память или нет
8 if (p2 != nullptr) {
9   *p2 = 60;
10 }
```

Оператор **delete** - возвращение блока динамической памяти обратно ОС, выделенной под одно значение конкретного типа

```
delete <переменная-указатель>;
```

```
1 int *p1;
2
3 p1 = new int;
4 *p1 = 89;
5
6 // Безопасно, даже если p1 == nullptr
7 delete p1;
```

Правило хорошего тона: для указателя на динамический блок памяти **обязателен** вызов оператора **delete**.

Указатели и динамическая память

Оператор **new[]**: выделение блока динамической памяти под массив заданного размера конкретного типа

```
(1) new <тип>[<размер_массива>];
(2) new (std::nothrow) <тип>[<размер_массива>];
```

```
1  int *p1 = new int[10];
2  p1[0] = 19;
3  p1[1] = -22;
4
5  size_t count;
6  get_value(count, "Введите размер массива: ");
7
8  int *dyn_array = new int[count];
9
10 for (size_t i = 0; i < count; i++) {
11   dyn_array[i] = i + 1;
12 }
```

Указатели и динамическая память

Оператор **new[]**: инициализация **числовых** массивов нулями

```
new <тип>[<размер>]{};
new (std::nothrow) <тип>[<размер>]{};
```

```
1  int *p1 = new int[10]{};
2  bool is_zero = p1[0] == 0;
3  // is_zero здесь равен true
4
5  size_t count;
6  get_value(count, "Введите размер массива: ");
7  double *dyn_reals = new double[count]{};
8
9  print("{");
10 for (size_t i = 0; i < count; i++) {
11   print(dyn_reals[i], ", ");
12 }
13 print("}\n");
```

Оператор **delete[]**: возвращение памяти, веделенной под массив

```
delete[] <переменная-указатель>;
```

```
1 int *my_ints = new int[10];
2
3 my_ints[2] = 98;
4 my_ints[4] = 89;
5
6 print("Сумма 3-го и 5-го элементов: ",
7       my_ints[2] + my_ints[4], "\n");
8
9 delete[] my_ints;
```

Пример - шаблон для работы с динамическим массивом

```
1  size_t count;
2  get_value(count, "Введите размер: ");
3
4  int *dyn_ints = new int[count];
5
6  for (size_t i = 1; i <= count; i++) {
7    if (i % 2 == 0) {
8      dyn_ints[i-1] = i * i;
9    } else {
10     dyn_ints[i-1] = -(i * i);
11   }
12 }
13
14 delete[] dyn_ints;
```

Указатели и динамическая память

Пример - как создать утечку памяти

```
1  double get_complex_rate(size_t count)
2  {
3    if (count == 0) {
4      return 0.0;
5    }
6
7    double *arr = new double[count], average;
8    // Сложная обработка массива
9    // и нет вызова оператора delete
10
11   return average;
12 }
13
14 get_complex_rate(5000000);
15 get_complex_rate(222000);
```

Пример - как устранить утечку памяти

```
1  double get_complex_rate(size_t count)
2  {
3    if (count == 0) {
4      return 0.0;
5    }
6
7    double *arr = new double[count], average;
8    // Сложная обработка массива
9    delete[] arr;
10
11   return average;
12 }
13
14 get_complex_rate(90000);
15 get_complex_rate(1500000);
```

Указатели и динамическая память

Многомерные указатели: указатели, как и массивы, могут иметь условную размерность. Она определяется количеством знаков * передназванием переменной. Например, общая форма создания двумерного указателя:

```
<тип> **<имя_переменной>;
```

Разбор такого набора звёзд делается справо налево:

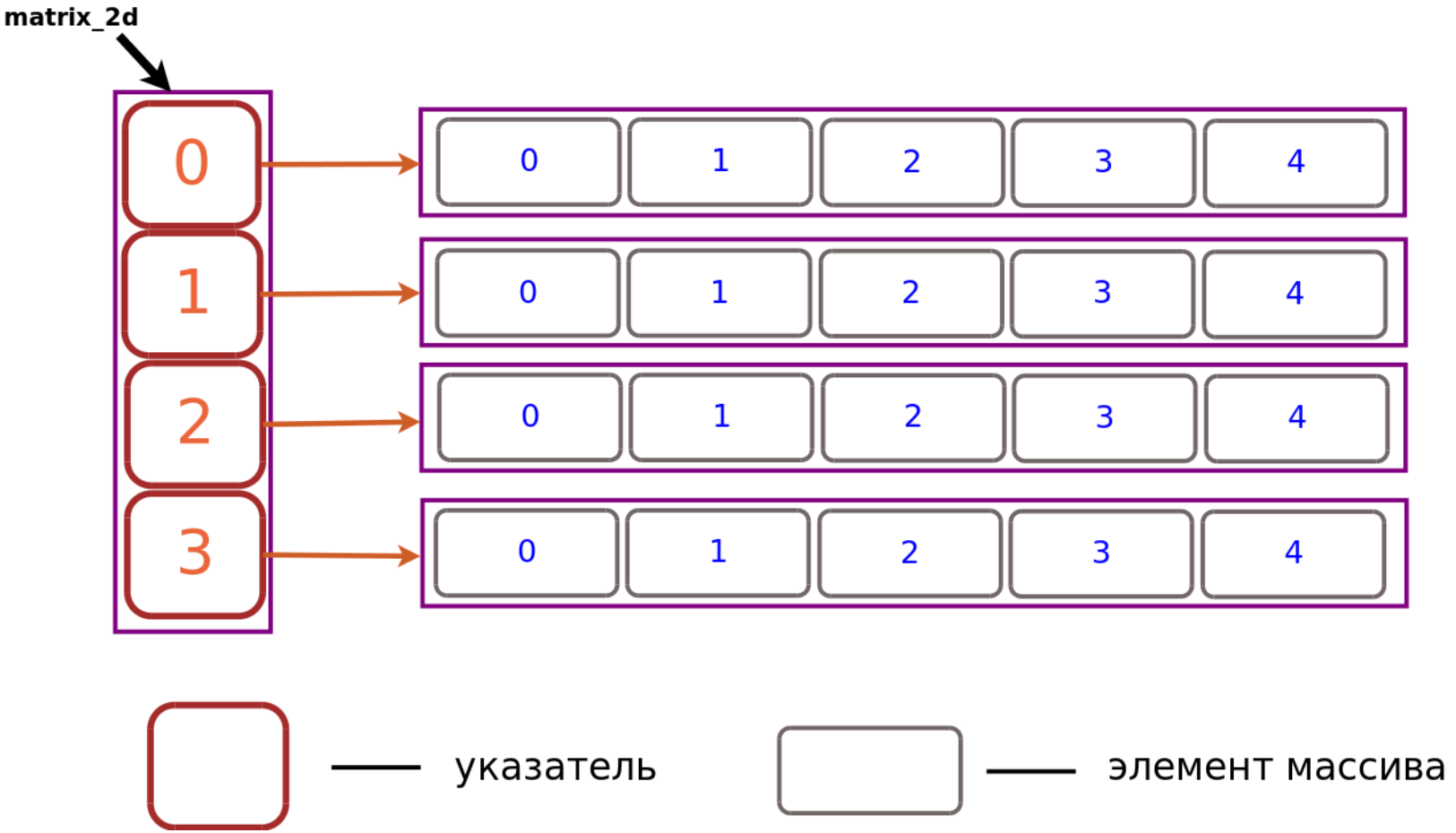
```
int **my_arr_2d;
```

my_arr_2d это указатель (первая звёздочка) на тип **int***, то есть - обычный одномерный массив других *указателей на int*. Если к этому массиву применяется оператор индексации вида **my_arr_2d[1]**, то получаем возможность работать с одиночным *указателем на int*.

Многомерные указатели: двумерный массив

```
1  size_t rows, cols;
2
3  print("Введите число строк и столбцов: ");
4  get_value(rows);
5  get_value(cols);
6
7  int **matrix_2d = new int*[rows];
8
9  for (size_t i = 0; i < rows; ++i) {
10   // Каждое разыменование многомерного
11   // указателя выкидывает один знак '*'
12   matrix_2d[i] = new int[cols];
13 }
```

Схематичное изображение двумерного массива 4x5 в C++



Многомерные указатели: работа с двумерным массивовм

```
1  for (size_t i = 0; i < rows; ++i) {
2    for (size_t j = 0; j < cols; ++j) {
3      matrix_2d[i][j] = rand_0_1_incl() + (i + j);
4    }
5  }
6  // работа с matrix_2d
7
8  // Возврат памяти в ОС
9  for (size_t i = 0; i < rows; ++i) {
10   // Удаление каждой строки
11   delete[] matrix_2d[i];
12 }
13 // Удаление массива указателей
14 delete[] matrix_2d;
```

Указатели и динамическая память

Для упрощения работы с двумерным массивом:

```
1  double** create_array_2d(size_t rows, size_t cols↩
       )
2  {
3    double **arr_2d = new double*[rows];
4    for (size_t i = 0; i < rows; i++) {
5      arr_2d[i] = new double[cols];
6    }
7    return arr_2d;
8  }
9
10 void delete_array_2d(double **&arr_2d,
11                       size_t rows, size_t cols)
12 {
13   for (size_t i = 0; i < rows; i++) {
14     delete[] arr_2d[i];
15   }
16   delete[] arr_2d;
17 }
```

И применение таких массивов:

```
1  size_t rows = 4, cols = 5;
2  double** my_matrix = create_array_2d(rows, cols);
3
4  print("Матрица ", rows, "x", cols, ":\n");
5  for (size_t i = 0; i < rows; ++i) {
6    for (size_t j = 0; j < cols; ++j) {
7      my_matrix[i][j] = rand_0_1_incl()
8                        + pow(0.5, i + j);
9      print(my_matrix[i][j], " ");
10   }
11   print("\n");
12 }
13 print("\n");
14
15 delete_array_2d(my_matrix, rows, cols);
```

Для самостоятельного разбора

Пример - изменение размера одномерного массива

```
1  double*** create_lattice_3d(size_t lx, size_t ly, size_t lz)
2  {
3    double ***arr_3d = new double**[lx];
4    for (size_t i = 0; i < lx; i++) {
5      arr_3d[i] = new double*[ly];
6      for (size_t j = 0; j < ly; j++) {
7          arr_3d[i][j] = new double[lz];
8      }
9    }
10   return arr_3d;
11 }
12
13 void delete_lattice_3d(double ***&arr_3d, size_t lx,
14                         size_t ly, size_t lz)
15 {
16   for (size_t i = 0; i < lx; i++) {
17     for (size_t j = 0; j < ly; j++) {
18       delete[] arr_3d[i][j];
19     }
20     delete[] arr_3d[i]
21   }
22   delete[] arr_3d;
23 }
```

Для самостоятельного разбора

Пример - изменение размера одномерного массива

```
1  #include <cstring>
2
3  int* re_size(int *p_arr, size_t cur_sz, size_t new_sz)
4  {
5    if (p_arr == nullptr) { return p_arr; }
6    if (cur_sz == new_sz) { return p_arr; }
7
8    size_t least_sz = cur_sz < new_sz ? cur_sz : new_sz;
9
10   int *p_new_arr = new int[new_sz];
11   memcpy(p_new_arr, p_arr, least_sz * sizeof(int));
12   delete[] p_arr;
13
14   return p_new_arr;
15 }
16
17 int *my_arr = new int[40];
18 //Расширить динамический массив my_arr до 65 элементов
19 my_arr = re_size(40, 65);
```