Лекция IV

27 октября 2017

Адрес переменной

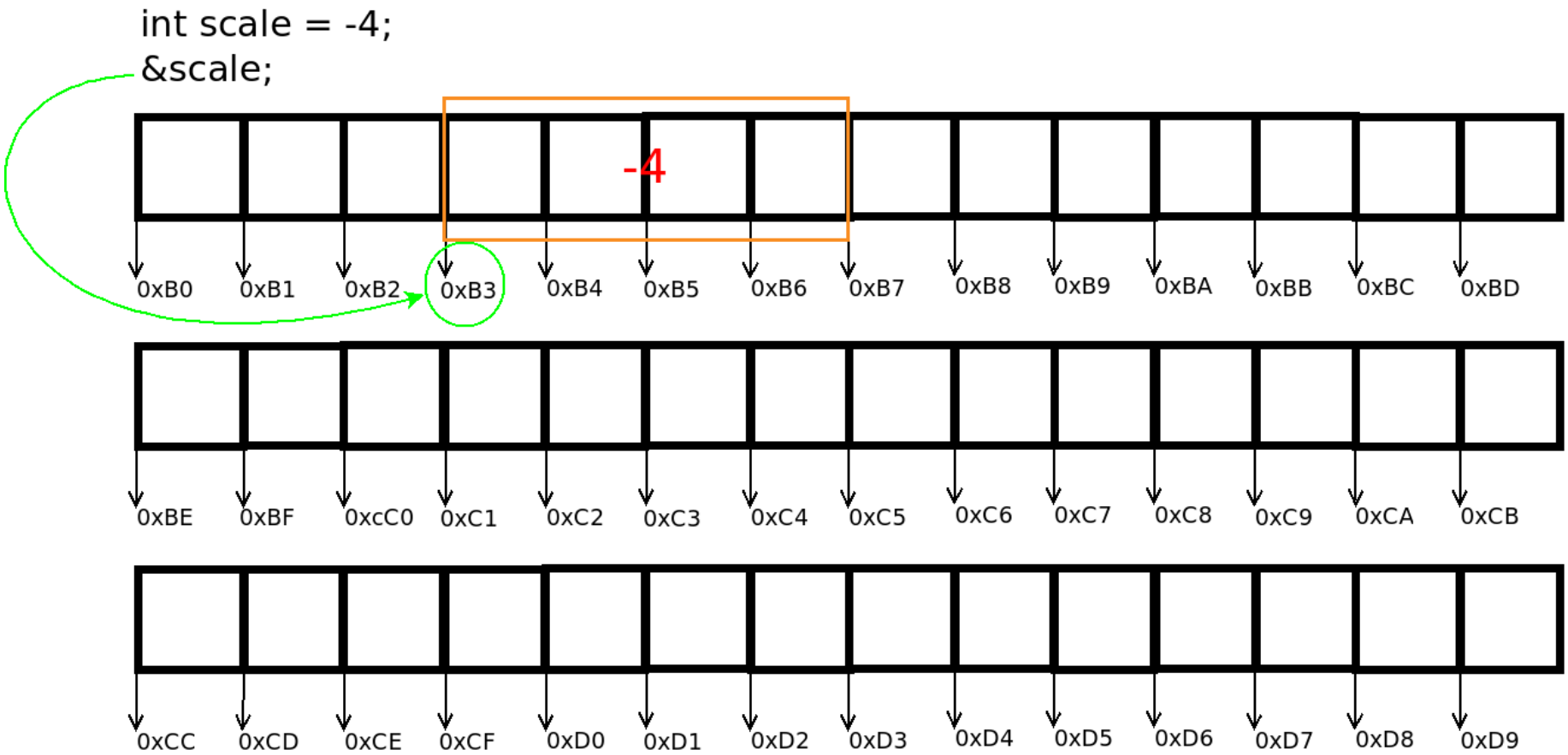
Каждая переменная любого типа в C++ связана с сопоставленным её блоком в оперативной памяти. Длина блока изменяется в байтах, для разных типов - различна.
Адресом переменной называют **номер первого байта** из блока, который отведён под неё. Это целое положительное число.
У переменной можно узнать адрес,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

```
1 int scale = -4;
2 double rate = 2.64;
3
4 cout << "Адрес rate: " << &rate;
5 cout << "\nАдрес scale: " << &scale;
```

Значения адреса выводятся на консоль в виде шестнадцатиричных чисел.

Адрес переменной

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



Указатели

Указателем - называют тип данных, переменные которого предназначены для хранения адресов других объектов (то есть, обыкновенных переменных простых, специальных или пользовательских типов). 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
```

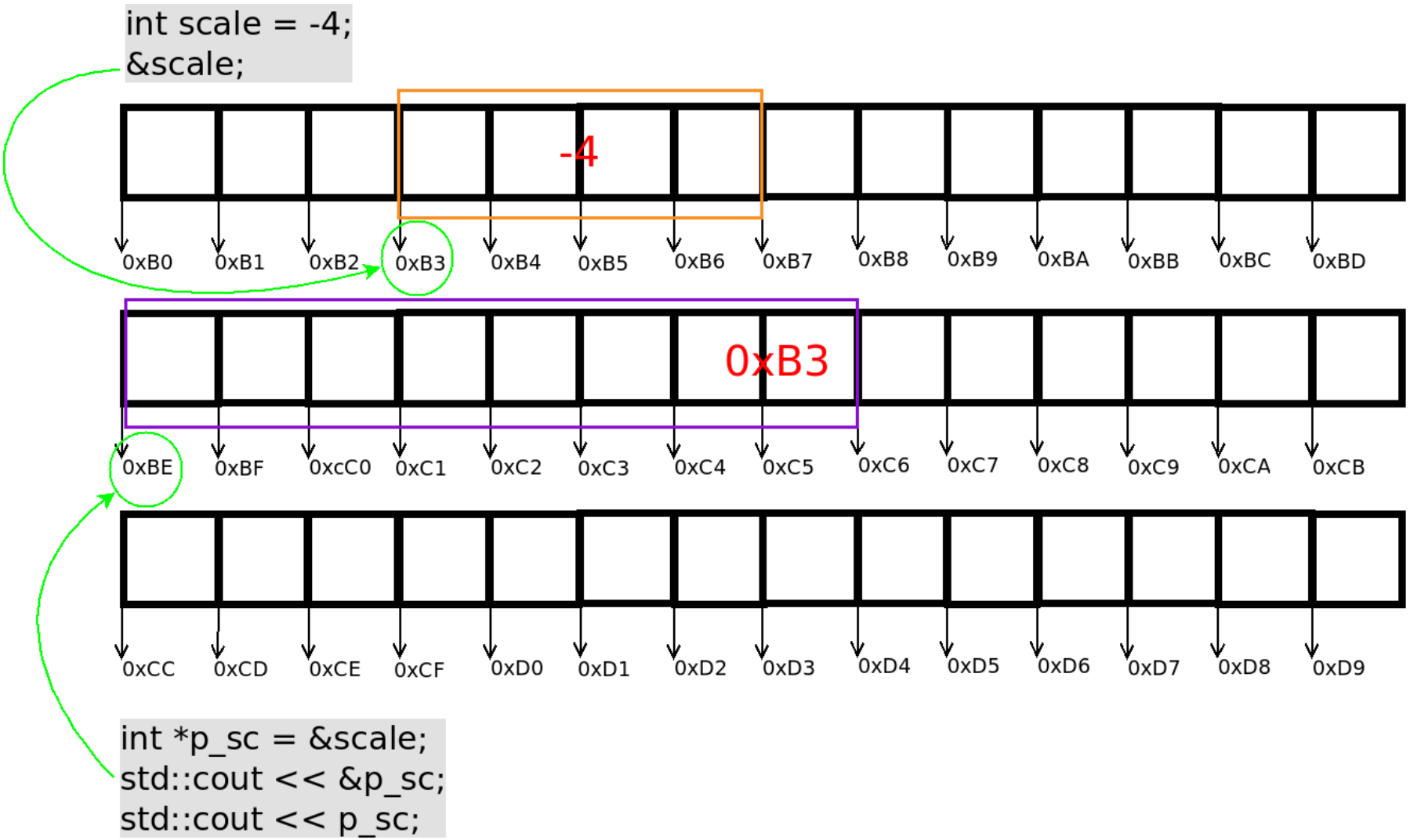
Операции с указателями: **присвоение значения**
Указателю может быть присвоено значение (адрес в памяти) либо с помощью операции взятия адреса у переменной, либо копированием значения из другого указателя.

```
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

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

```
1 // В языке C для нулевой адрес определён как NULL
2 int *p2 = nullptr;
3
4 // Операции с p2 ...
5 if (p2 != nullptr) {
6   // что-нибудь делаем
7 }
```

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 значения переменной, адрес которой сохранён в указателе.
Синтаксис

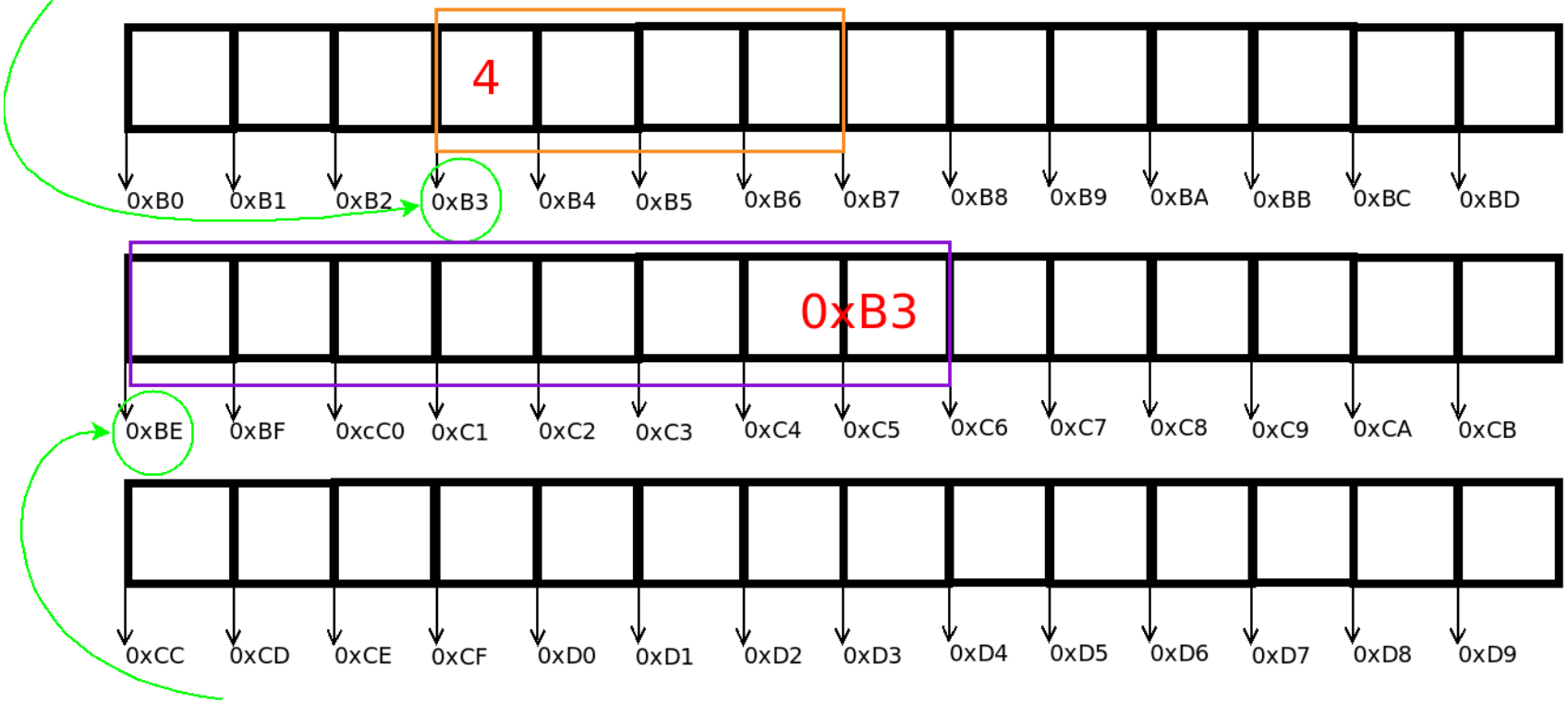
```
*<имя_переменной_указателя>;
```

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

Указатели

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 **nullptr**.

Так никогда не делать!

```
1 double *p_real;
2 cout << "Что за число тут: " << *p_real;
3
4 int *p_int = nullptr;
5 cout << "А вдруг сработает: " << *p_int;
```

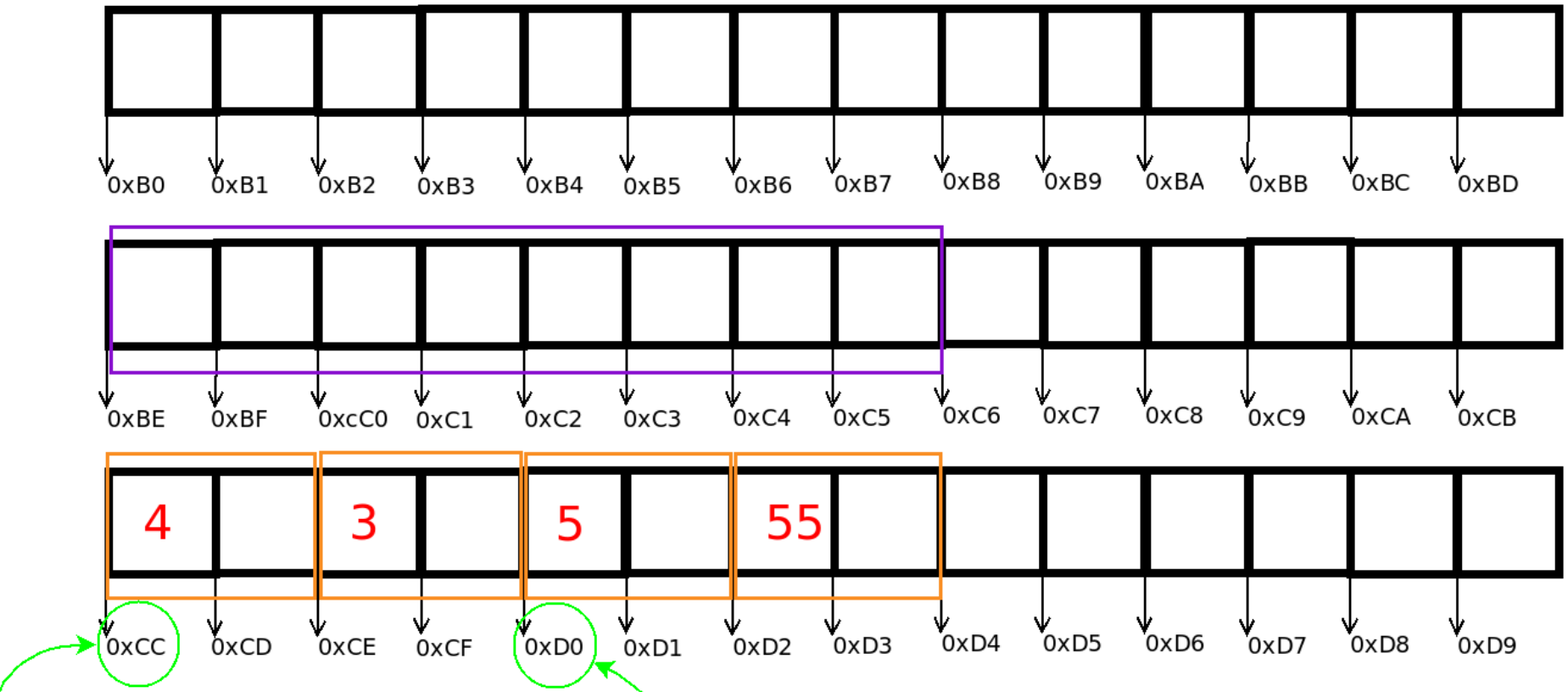
Операции с указателями: **разыменование**

```
1  void c_swap(int *i1, int *i2)
2  { // Как делать в стиле C
3    if (i1 != nullptr && i2 != nullptr) {
4      int tmp = *i1; *i1 = *i2; *i2 = tmp;
5    }
6  }
7
8  void cpp_swap(int& i1, int& i2)
9  { // Как в C++
10   int tmp = i1; i1 = i2; i2 = tmp;
11 }
12
13 int n1 = 15, n2 = 103;
14 c_swap(&n1, &n2); cpp_swap(n1, n2);
```

Стоит отметить, что сами переменные-указатели передаются в функции **по значению**. Здесь отличий от переменных обычных типов нет.

Указатели и массивы

Вспомним, как массив располагается в памяти




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

&vec[0];

&vec[2];
```

Указатели и массивы

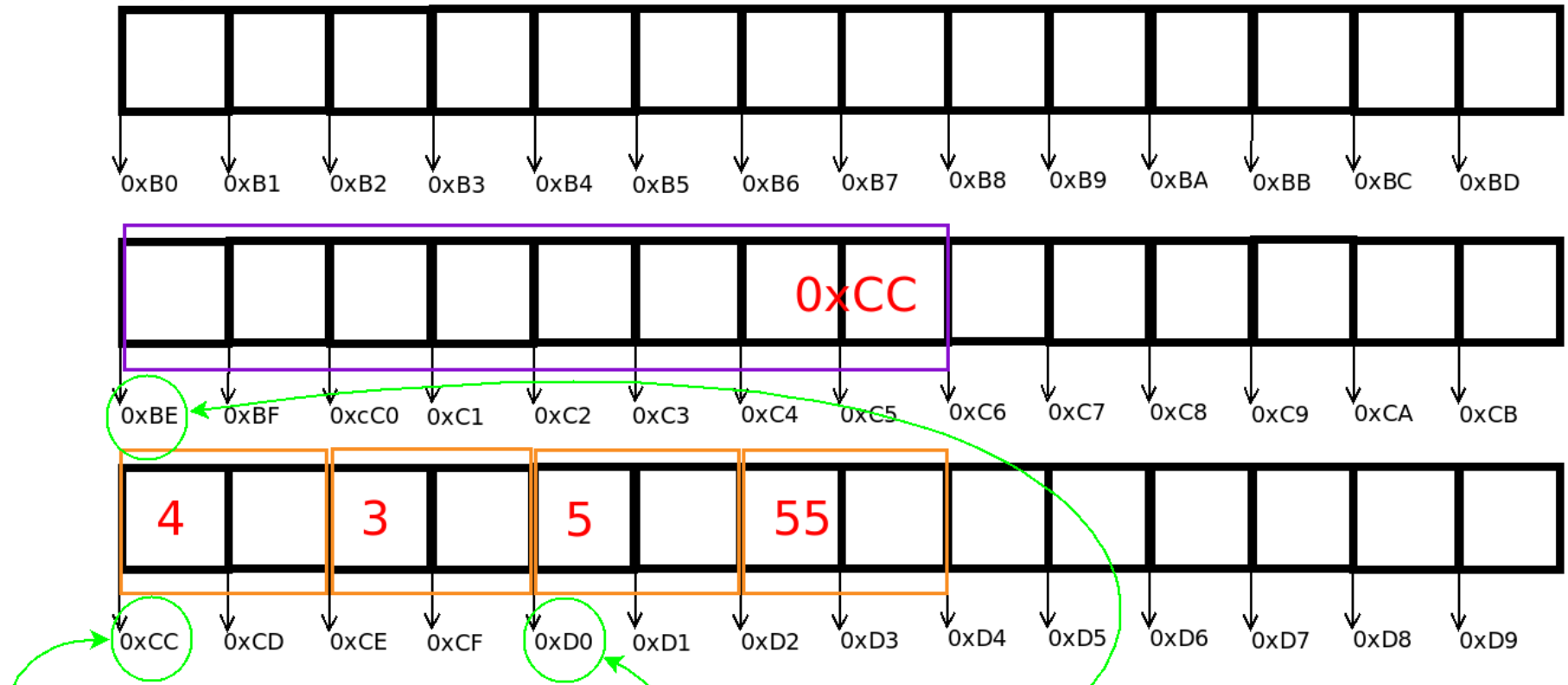
Связь указателей и массивов

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

```
1 void print_array(short* arr, size_t count);
2 ...
3 short vec[4] = {4, 3, 5, 55};
4
5 print_array(vec, 4);
```

Указатели и 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;
```

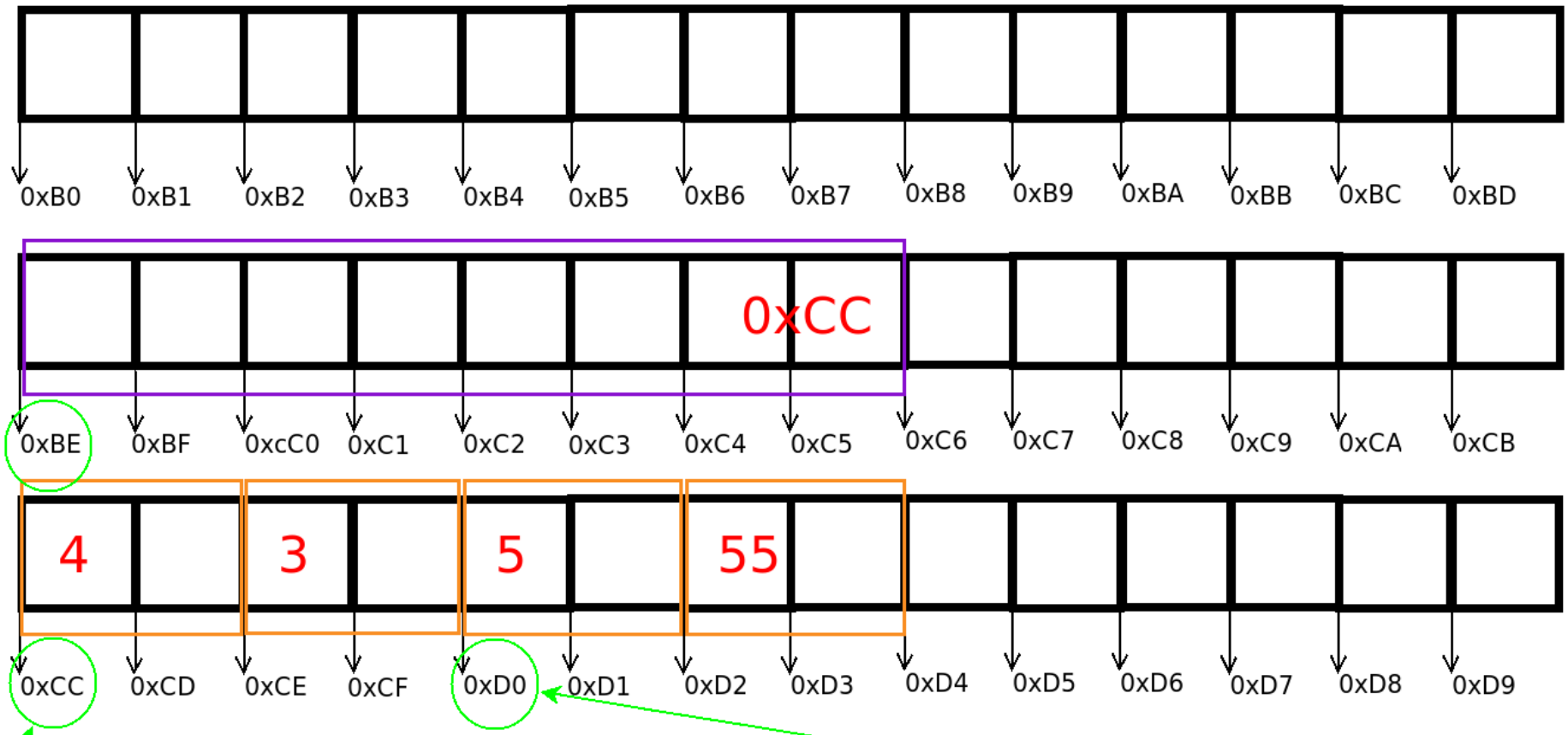
Указатели и массивы

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 cout << "Значение третьего элемента: "
6      << *(p_vec + 2) << "\n";
```

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

Сложение указателя с целым числом графически




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

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

Операции с указателями: **сложение с целым числом**.
Как видно из примеров, особенно полезно сложение при использовании указателя для работы с элементами массива.

```
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 массивы

Операции с указателями: **индексация**

Индексация указателя выполняет два действия:

1. Сместиться на количество блоков, равных индексы, от текущего адреса
2. Получить значение по адресу, **после смещения**

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

Указатели и массивы

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

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

Указатели

Указатель на тип **void**.

1. Указателю на **void** может быть присвоено значение любого другого указателя
2. Указатель на **void** может быть присвоен любой другой указатель **только с использованием явного приведения типа**.
3. Для указателей на **void** запрещены операции **разыменования**, **индексации** и **вычитания указателей**.

```
1 short vec[4] = {4, 3, 5, 55};
2 short *p1 = vec, *p2;
3
4 void *pv1 = p1;
5
6 // Явно приводим тип указателя-на-void
7 // к указателю на short
8 p2 = static_cast<short*>( pv1 );
9 cout << "Первый элемент vec: " << *p2 << "\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 **байтах**
- Динамический блок памяти существует до тех пор, пока не будет вызван оператор для его возвращения в ОС
- 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). В задачах вычислительной физики - особенно.

Указатели и динамическая память

Оператор **new** - запрос **одного** блока динамической памяти у ОС конкретного типа данных.

```
(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 }
```

Указатели и динамическая память

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

```
new <тип_данных>[<размер>];
new (nothrow) <тип_данных>[<размер>];
```

```
1  int *p1, *p2;
2  p1 = new (nothrow) int[10];
3
4  if (p1 != nullptr) {
5    p1[0] = 19;
6  }
7
8  int count = 6;
9  p2 = new int[count];
10 p2[2] = 3;
```

Указатели и динамическая память

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

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

```
1 int *p1;
2 p1 = new int[10]{};
3
4 bool is_zero = p1[0] == 0;
5 // is_zero здесь равен true
6
7 int count = 6;
8 double *p2;
9 p2 = new double[count]{};
```

Указатели и динамическая память

Оператор **delete** - возращение блока динамической памяти обратно ОС, выделенной под одно значение

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

```
1  int *p1;
2  p1 = new (nothrow) int;
3
4  if (p1 != nullptr) {
5    *p1 = 89;
6    // ...
7  }
8
9  // Безопасно, даже если p1 == nullptr
10 delete p1;
```

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

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

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

```
1 int *p1;
2 p1 = new int[10];
3
4 p1[4] = 89;
5 p1[2] = 98;
6 cout << "Сумма 3-го и 5-го элементов: "
7      << p1[2] + p1[4] << "\n";
8
9 delete[] p1;
```

Пример - заполнение массива квадратом индекса

```
1  int *dyn_arr;
2  size_t count;
3  cout << "Введите размер: ";
4  cin >> count
5
6  dyn_arr = new int[count];
7
8  for (size_t i = 0; i < count; ++i) {
9    dyn_arr[i] = i * i;
10 }
11
12 delete[] dyn_arr;
```

Пример - как создать утечку памяти

```
1  double get_complex_average(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_average(50000000);
15 get_complex_average(222100);
```

Пример - как устранить утечку памяти

```
1  double get_complex_average(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_average(90000);
15 get_complex_average(1500000);
```

Указатели и динамическая память

Многомерные указатели: указатели, как и массивы, могут иметь условную размерность. Она определяется количеством знаков * передназванием переменной.

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

Пример - многомерные указатели

```
1  //... см. предыдущий слайд
2  for (size_t i = 0; i < rows; ++i) {
3    for (size_t j = 0; j < cols; ++j) {
4      matrix[i][j] = rnd_0_1() + (i + j);
5    }
6  }
7  // работа с matrix
8
9  // Возврат памяти в ОС
10 for (size_t i = 0; i < rows; ++i) {
11   // Удаление каждой строки
12   delete[] matrix[i];
13 }
14 // Удаление массива указателей
15 delete[] matrix;
```

Указатели и динамическая память

Пример - изменение размера выделенного блока конкретного типа (ещё и указатель по ссылке передаём)

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

Динамический массив

Динамический массив представлен в C++ типом **vector**. Он определён в общей (шаблонной) форме для всех других типов. Для его использования следует подключить следующий заголовочный файл:

```
1 #include <vector>
```

Динамический массив

Динамический массив: создание переменных, общая форма:

```
1 // (1)
2 vector<Type> var1
3
4 // (2)
5 vector<Type> var2{size_t count}
6
7 // (3)
8 vector<Type> var2{size_t count, Type value}
```

1. **(1)** - создаёт массив нулевой длины. Память под элементы не выделяется.
2. **(2)** - создаём массив и выделяем место под **count** элементов. Начальные значения элементам не присваиваются.
3. **(3)** - создаём массив под **count** элементов и **каждому из них** присваиваем значение **value**.

Динамический массив

Динамический массив: создание переменных, примеры:

```
1 vector<int> int_array;
2
3 vector<double> real_array{10};
4
5 string base_value = "ABC";
6 vector<string> str_array{5, base_value};
7
8 // Можно делать и так:
9 vector<int> int_arr2 = {1, 5, 6, 7, 8, 10};
```

Динамический массив

Динамический массив: методы для работы с количеством элементов

```
vector<Type> my_arr(10);

(1) size_t my_arr.size();
(2) size_t my_arr.max_size();
(3) bool my_arr.empty();
```

1. **(1)** - узнать текущий размер массива
2. **(2)** - узнать потенциально максимальное количество элементов
3. **(3)** - метод возращает **true** если массив не содержит ни одного элемента, **false** - в противоположном случае

Динамический массив

Динамический массив: методы для работы с количеством элементов

```
vector<Type> my_arr(10);

(4) void my_arr.resize(size_t new_size);
    void my_arr.resize(size_t new_size, type val);
(5) void my_arr.reserve(size_t count);
(6) void my_arr.clear();
```

4. **(4)** - поменять размер массива на **new_size**. Если **new_size** меньше текущего размера - лишние элементы удаляются. Если больше - то выделяется память под нужное количество элементов. С помощью **val** - добавляемым элементам можно задать конкретное начальное значение
5. **(5)** - если **count** больше текущего размера массива, под недостающие элементы выделяется память
6. **(6)** - удалить все элементы из массива

Динамический массив

Динамический массив: методы для работы с количеством элементов, примеры

```
1  vector<int> int_arr, int_arr2(14, 5);
2
3  string base_value = "ABC";
4  vector<string> str_arr{5, base_value};
5
6  cout << "\nРазмер str_arr: " << str_arr.size();
7  cout << "\nint_arr пуст? " << int_arr.empty();
8
9  str_arr.resize(10, "mmm");
10 cout << "\nРазмер str_arr: " << str_arr.size();
11
12 int_arr2.reserve(20);
13 cout << "\nРазмер int_arr2: " << int_arr2.size();
```

Динамический массив

Динамический массив: методы для доступа к элементам

```
(1) Type& my_arr[size_t n];
(2) Type& my_arr.at(size_t n);
```

1. **(1)** - получить ссылку на элемент с индексом **n**
2. **(2)** - получить ссылку на элемент с индексом **n** и вызвать остановку программы при неправильном индексе

```
1 vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,↩
      10};
2
3 int_arr[0] = 8;
4 cout << "\nПервый элемент равен: " << int_arr[0];
5 int_arr.at(3) = 14;
6
7 cout << "\nЧетвёртый: " << int_arr.at(3);
8 // Поведение неопределено:
9 cout << "\nНеизвестный: " << int_arr[3001];
```

Динамический массив

Динамический массив: методы для доступа к элементам

```
(3) Type& my_arr.front();
(4) Type& my_arr.back();
```

3. **(3)** - получить ссылку на первый элемент
4. **(4)** - получить ссылку на последний элемент

```
1  vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,↩
       10};
2
3  cout << "Первый элемент: " << int_arr.front();
4  cout << "Последний элемент: " << int_arr.back();
5
6  int_arr.front() = 25;
7  int_arr.back() += 10;
8
9  cout << "Первый элемент: " << int_arr.front();
10 cout << "Последний элемент: " << int_arr.back();
```

Динамический массив

Динамический массив: методы для доступа к элементам

```
(5) Type& my_arr.push_back(Type& value );
(6) void my_arr.pop_back();
```

5. **(5)** - добавить элемент **value** в конец массива
6. **(6)** - удалить последний элемент

```
1 vector<int> int_arr = {1, 2, 3, 4, 5, 6, 7, 8, 9,↩
      10};
2
3 int_arr.push_back(888);
4 int_arr.push_back(777);
5 cout << "Последний элемент: " << int_arr.back();
6
7 int_arr.pop_back();
8 cout << "Последний элемент: " << int_arr.back();
```