IV

14 марта 2020

Адреса, переменные и память
Углублённое расследование

Адрес переменной (повторение)

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 printf("Адрес rate: %p \n", &rate);
5 printf("Адрес scale: %p \n", &scale);
```

Возможный вывод:

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

Адрес переменной (повторение)

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

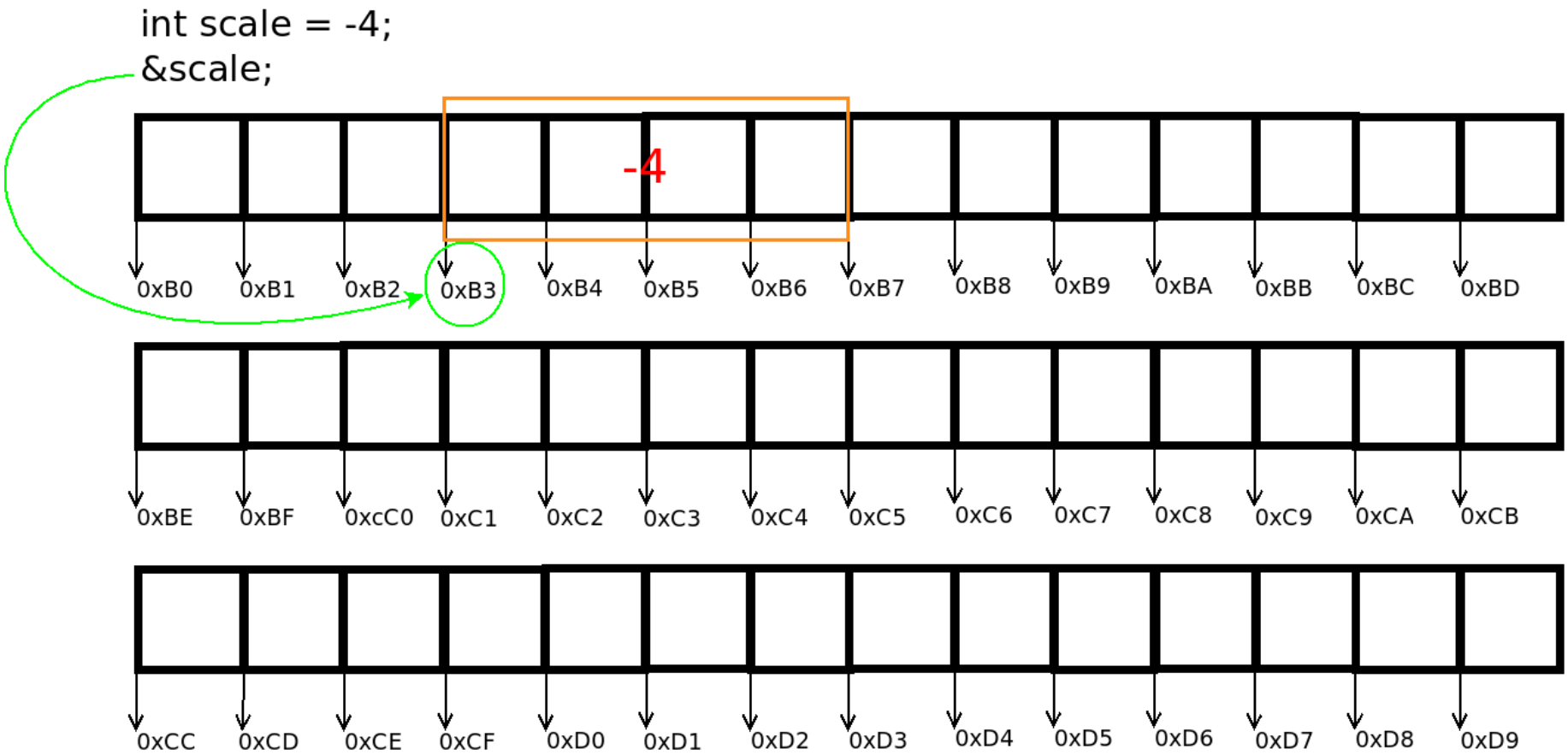
```
1 int scale = -4;
2 double rate = 2.64;
3
4 printf("Size of int: %zu\n", sizeof(int));
5 printf("Size of double: %zu\n", sizeof(rate));
6
7 int arr[14];
8 size_t arr_size = sizeof(arr) / sizeof(arr[0]);
9 printf("array has %zu elements\n", arr_size);
```

Возможный вывод:

```
Size of int: 4
Size of double: 8
array has 14 elements
```

Адрес переменной (повторение)

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 манипуляций 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 манипуляций 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

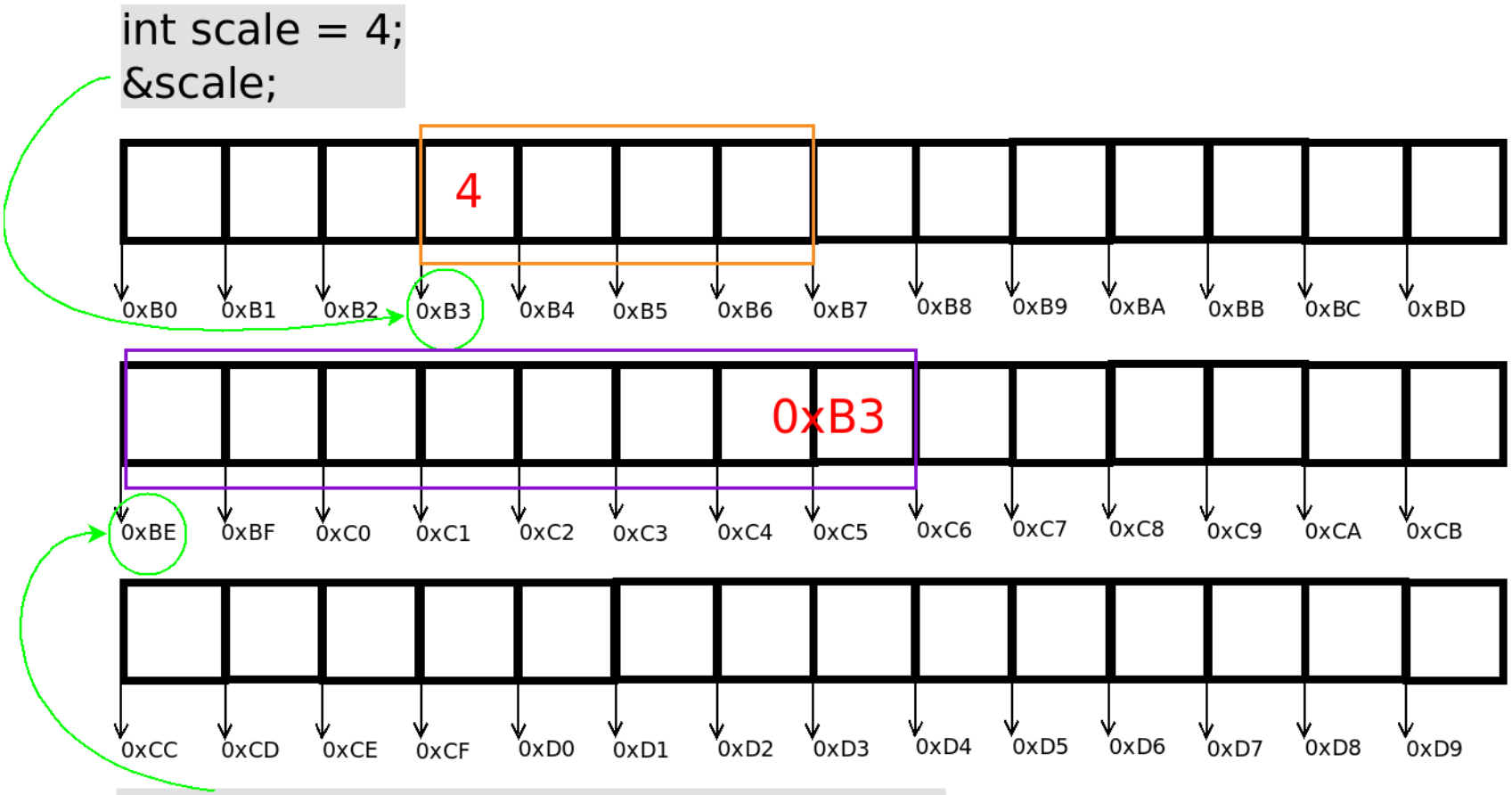
Операции с указателями: **присвоение значения «=»**
Указателю может быть присвоено значение (адрес в памяти) либо с помощью операции взятия адреса у переменной, либо копированием значения из другого указателя.

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

1-я строка: определяем указатель на целое **p_sc** и переменную целого типа **scale**.
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.



Размер переменной указателя

Независимо от того, какого типа указатель, количество байт выделяемых под *переменную-указатель* всегда одинаково. Этот размер совпадает с размером типа **size_t**.

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 типа **nullptr_t**. 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   printf("All right, continue\n");
7 }
```

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 типа **nullptr_t**. 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   printf("All right, continue\n");
7 }
```

Правило для работы с указателями: переменная-указатель **всегда** должна быть *инициализирована*. Ей надо или присвоить реальный адрес переменной, или *нулевой адрес*.

Когда C++ гарантирует нулевое значение

Если переменная-указатель объявлена в глобальной области видимости и явно не инициализирована, она получает значение равное нулевому указателю **nullptr**.

Указатели

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

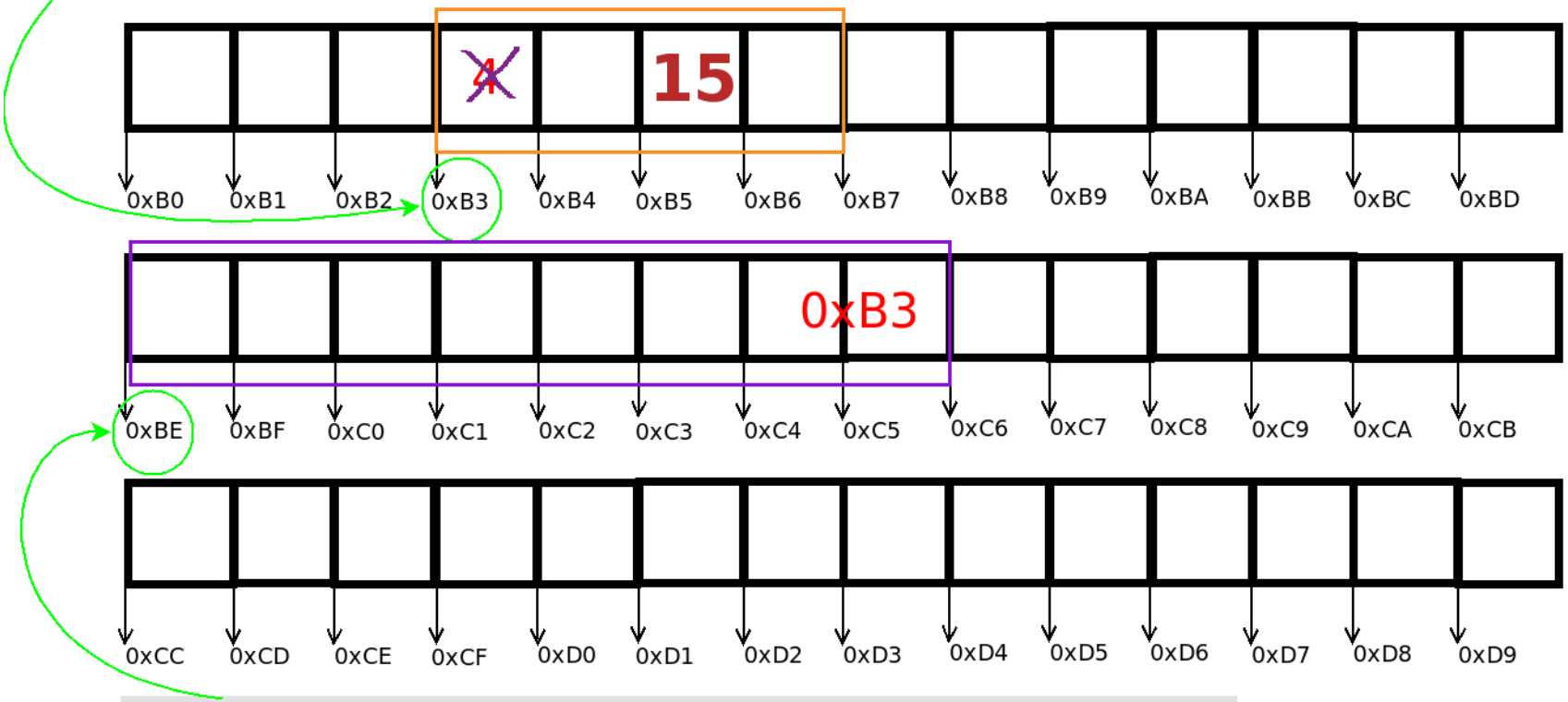
`*<имя_переменной_указателя>;`

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

Указатели

Схематично разыменование можно показать так:

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




```
int *p_sc = &scale;
printf("Value of scale using pointer: %d\n", *p_sc);
*p_sc = 15;
printf("After assigment: %d", *p_sc);
```

Операции с указателями: **разыменование**

Предупреждение: разыменование не инициализированной 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 printf("What's number here: %f\n", *p_real);
3
4 int *p_int = nullptr;
5 printf("We are lucky (probably): %d\n",
6         *p_int);
```

Указатели как параметры функций

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

- при передаче указателя в функцию **по значению** создаётся отдельная переменная-указатель, в неё копируется адрес передаваемого;

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

- для передачи указателя **по ссылке** указывается знак амперсанда «**&**» после символа «*****»;

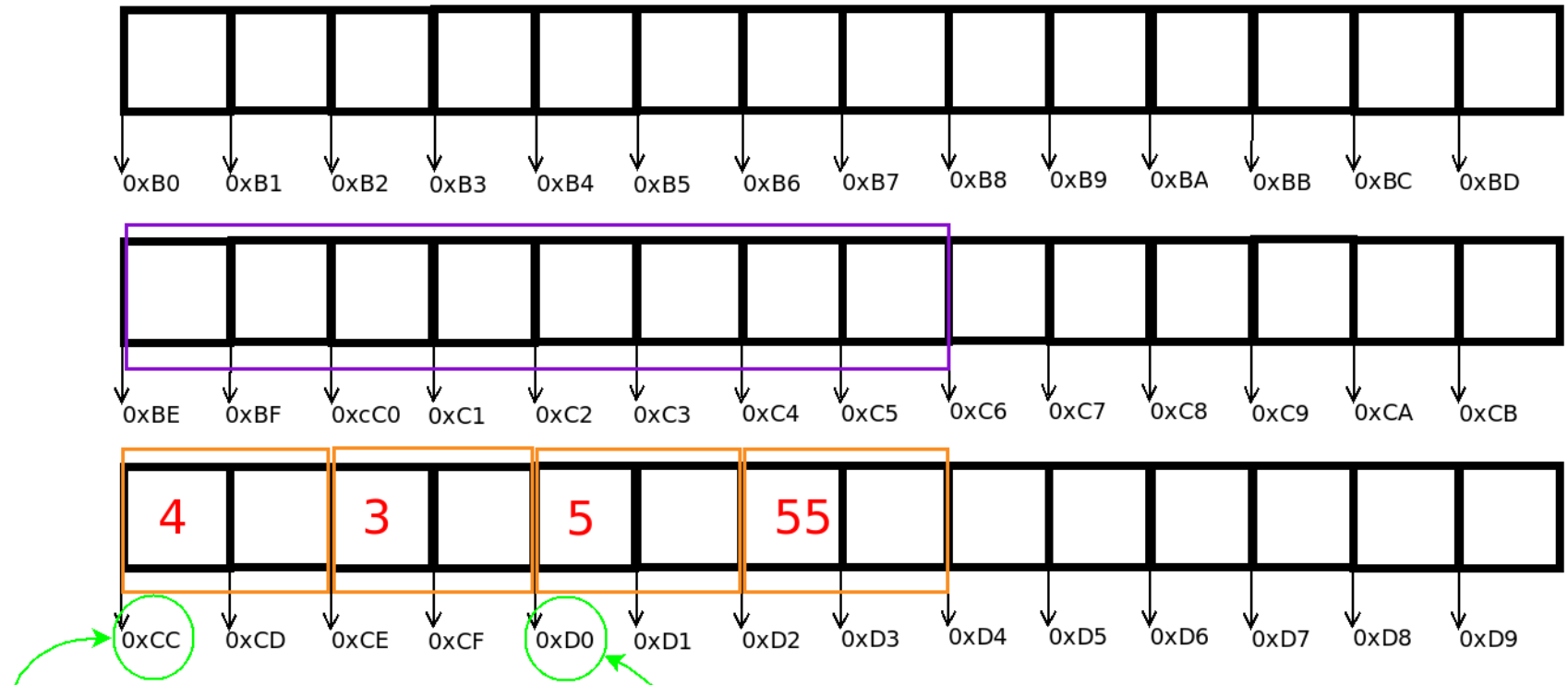
```
5 void pass_pointer_by_ref(int *& pointer);
```

- кроме того, с помощью оператора взятия адреса «**&**» из переменной можно «сделать» указатель.

```
6 pass_pointer_by_value( &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

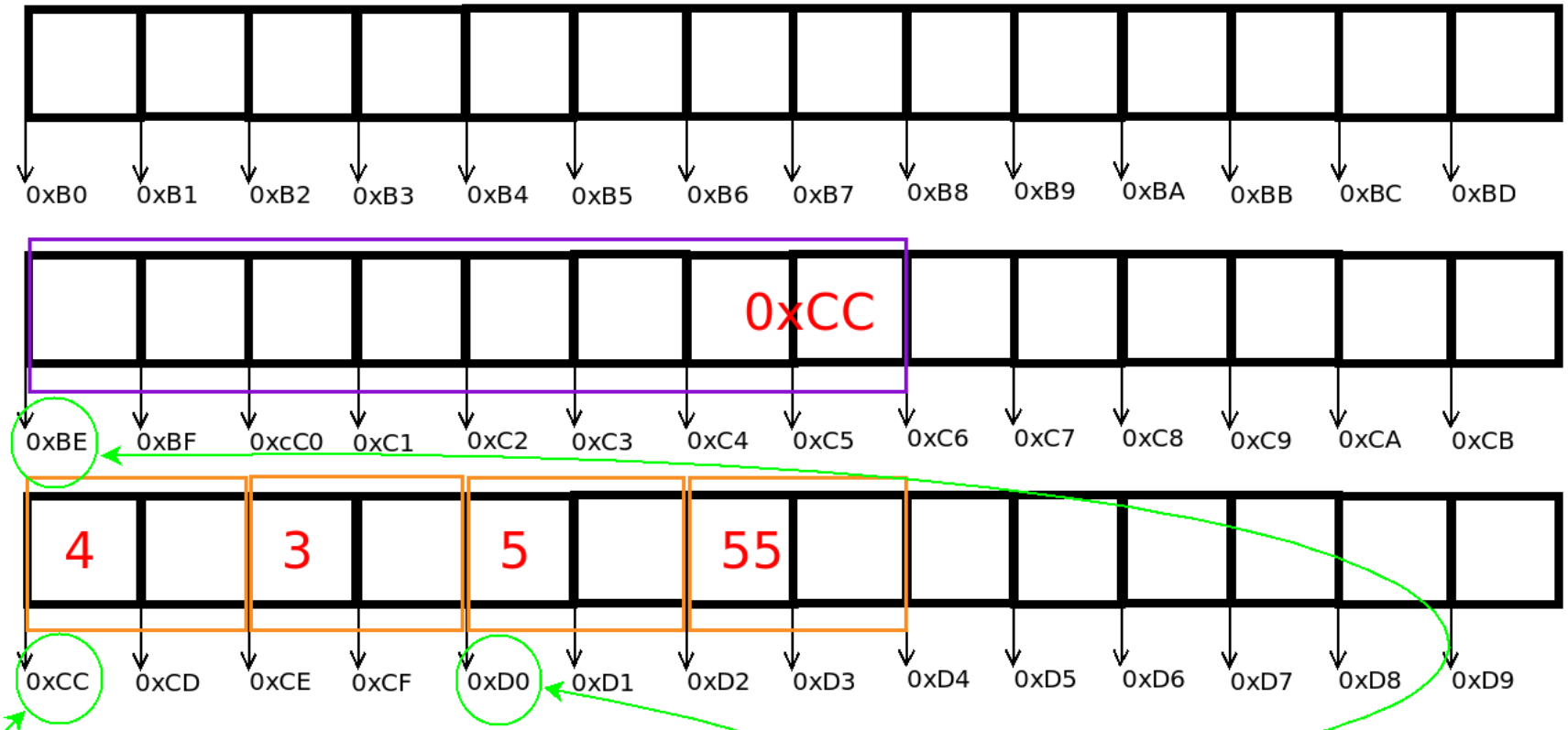
Сходства и различия указателей и массивов

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

```
1 // С прошлой лекции
2 void show_array(int* arr, size_t count);
3 ...
4 int vec[4] = {4, 3, 5, 55};
5 show_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;

printf("Число: %d", *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 $\mathrm{n} * \mathrm{sizeof}(< \mathrm{type} >)$ байт (вправо или влево - зависит от знака **n**).

Указатели и статические массивы

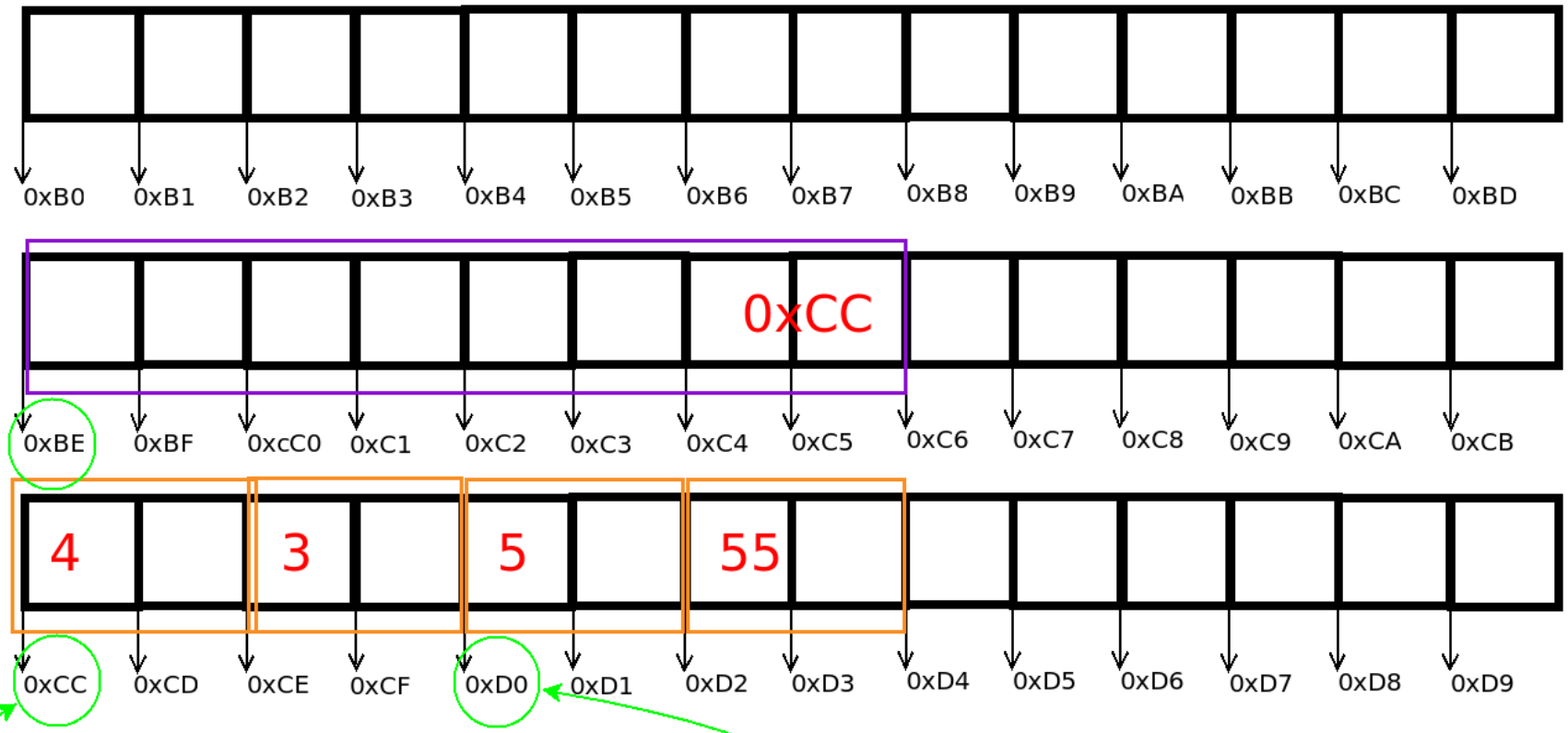
Операции с указателями: **сложение с целым числом**: результатом операции прибавления целого числа **n** к указателю является новый указатель, значение которого смещено на $n * \text{sizeof}(< \text{type} >)$ байт (вправо или влево - зависит от знака **n**).

```
1 short vec[4] = {4, 3, 5, 55};
2 short *p_vec = vec;
3
4 // Печатаем 5
5 printf("Third element: %d\n",
6         *(p_vec + 2),"\n");
```

Смещение происходит блоками, размер которого определяется типом указателя (указатель на **int**, указатель на **double**, указатель на **char** и другие 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. *Разыменовать* полученный в результате смещения адрес

```
1 int vec[4] = {4, 3, 5, 55};
2 int *p_vec = vec;
3
4 if (p_vec[2] == *(p_vec + 2)) {
5   printf("Values are equal\n");
6 }
7
8 printf("Forth element: %d\n", *(vec + 3));
```

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.

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  printf("There are %d elements ",
7         "in range [first; last) \n", diff);
8
9  diff = p1 - p2;
10 // Печатает -3
11 printf("Inverse order: %d\n", diff);
```

Константные указатели

```
1 int my_number = 1331, scale = 88;
```

Переменные-указатели могут быть объявлены как константные. Можно выделить три случая:

1. **Указатель на константную переменную**: нельзя менять значение переменной через указатель

```
2 const int *p1 = &my_number;
3 *p1 = 132; // !Ошибка компиляции!
4 p1 = &scale; // Всё ок, поменяли адрес
```

2. **Константный указатель**: нельзя менять адрес, сохранённый в указателе

```
5 int * const p2 = &my_number;
6 *p2 = 132; // Всё ок
7 p2 = &scale; // !Уже не получится!
```

3. **Константный указатель на константную переменную**: выполняются оба предыдущих условия

```
8 const int * const p3 = &my_number;
9 *p3 = 132; // !Ошибка компиляции!
10 p3 = &scale; // !Снова ошибка!
```

Указатели и динамическое управление памятью

Формализация инициализации I

Инициализация **локальных** переменных.

- *Инициализация по умолчанию* (default initialization)

```
1 int value_n;
2 int cookies[10];
```

- инициализация начальным значением
  - *Инициализация значением по умолчанию* (value initialization)

```
3 int value_n{};
4 int *ptr{}; // Значение в ptr -> nullptr
```

Для фундаментальных типов и указателей — происходит присвоение нулевых значений.

  - *Прямая инициализация* (direct initialization)

```
5 int value_n{101};
6 int *ptr{ &n };
```

Формализация инициализации II

- *Инициализация копированием* (copy initialization)

```
7 int value_n = 255552;
8 int *ptr = &n;
9 int *other = nullptr
```

- *Инициализация составных типов* (list initialization)

```
10 int happy_digits[] = {5, 8, 7, 4, 9};
11 int other_digits[10] {1, 2, 3};
```

Для **глобальных** переменных, к которым в явном виде не применяется никакая из форм инициализации, вызывается *инициализация значением по умолчанию*.

Указатели и динамическая память

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

- Язык C++ предоставляет операторы для создания объектов (в смысле C++)/массива объектов, время жизни которых определяется **вручную**
- Операторы выделяют *динамический* блок памяти под нужное количество объектов и применяют к каждому элементу **инициализацию**
- Динамический блок памяти принадлежит запущенной программе до тех пор, пока не будет «удалён». Под *удалением* подразумевается либо вызов соответствующего оператора для возвращения памяти в ОС, либо завершение работы программы.
- Доступ к динамическим блокам осуществляется **только** при помощи указателей

Указатели и динамическая память

Оператор **new** - запрос динамической памяти у ОС на один объект заданного типа. Размер выделенного блока совпадает с размером типа данных. При успешном выполнении, оператор возращает **адрес выделенного блока**.

```
new <тип>{<значение_для_инициализации>};
new <тип>(<значение_для_инициализации>);
```

```
1  int *p1 = new int; // инициализация по умолчанию
2  *p1 = 89;
3  printf("Value of dynamic int obj is %d\n", *p1);
4
5  // инициализация значением по умолчанию
6  int *p2 = new int{};
7  printf("Value pointed by p2 is %d\n", *p2);
8
9  int *p3 = new int{101}; // прямая инициализация
10 bool is_101 = *p3 == 101;
```

При ошибке выделения: завершение работы программы

Оператор **delete** - возращение блока динамической памяти обратно ОС, выделенной под один объект конкретного типа

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

```
1 int *p1 = new int;
2 *p1 = 89;
3 delete p1;
4 // !Недопустимо, ошибка времени выполнения!
5 //delete p1;
6
7 p1 = nullptr;
8 delete p1; // Безопасно
9 delete p1; // Сколь угодно раз подряд
```

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

```
new <тип>[<размер_массива>] {<инициализация>};
```

```
1 size_t elems_count;
2 printf("Enter array size: ");
3 scanf("%zu", &elems_count);
4
5 int *dyn_array = new int[elems_count];
6 for (size_t i = 0; i < elems_count; i++) {
7   dyn_array[i] = i + 1;
8 }
```

Выделяется массив под заданное с консоли количество элементов, к массиву применяется *инициализация по умолчанию*. Затем происходит работа с ним.

Оператор **new[]**: инициализация массивов начальными значениями

```
1 int *arr1 = new int[10]{};
2 bool is_zero = p1[0] == 0;
3 // is_zero здесь равен true
4
5 double *arr2 = new double[12] {1.0, 2.0, 3.0};
6 is_zero = arr2[4] == 0.0;
7 // опять true
```

Указатели и динамическая память

Оператор **delete[]**: возращение 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  printf("Sum of third and fifth is %d\n",
7         my_ints[2] + my_ints[4]);
8
9  delete[] my_ints;
10 // delete[] my_ints; // !Так не делать!
```

Повторение: безопасно применять операторы **delete**/**delete[]** можно только к указателю, который хранит *нулевой адрес* (значение **nullptr**).

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

```
<тип> **<имя_переменной>;
```

Разбор такого набора звёзд делается справо налево:

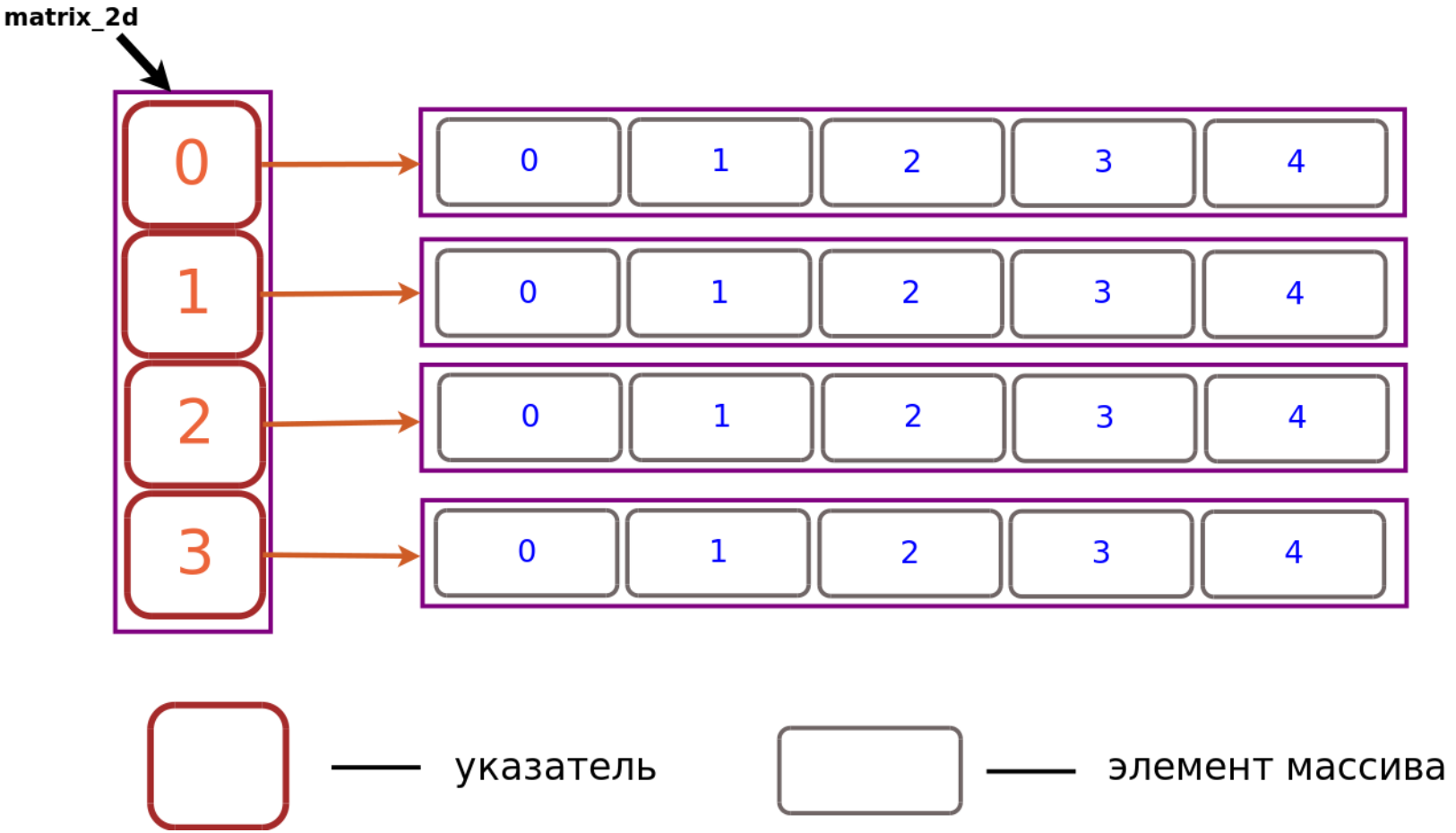
```
int **my_arr_2d;
```

my_arr_2d это указатель (первая звёздочка справа) 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("Enter cols and rows numbers: ");
4  scanf("%zu %zu", &rows, &cols);
5
6  int **matrix_2d = new int*[rows];
7
8  for (size_t i = 0; i < rows; ++i) {
9    // Каждое разыменование многомерного
10   // указателя выкидывает один знак '*'
11   matrix_2d[i] = new int[cols];
12 }
```

Схематичное изображение двумерного массива 4x5 в C++



Многомерные указатели: работа с двумерным массивом

```
1  for (size_t i = 0; i < rows; ++i) {
2    for (size_t j = 0; j < cols; ++j) {
3      matrix_2d[i][j] = rand_a_b() + (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 в C++. Резюме

- Манипулируют адресами в оперативной памяти.
- Арифметические операции работают по отдельным правилам, отличным от операций с целыми числами.
- Доступен оператор индексации
- Всегда следует инициализировать указатели.
- Нулевой адрес — **nullptr**.
- Являются единственным способом в C++ работать с динамическими массивами различных размерностей без привлечения стандартной библиотеки.
- Значение указателя может быть возвращено из функций.
- Выделение и высвобождение динамической памяти должно находиться в одной области видимости.

Динамические массивы: вспомогательные функции

Если в процессе написания кода программы возникает необходимость создания больше одного динамического массива, неплохим решением будет вынос подзадач по выделению/высвобождению памяти в отдельные функции (та самая *декомпозиция*).
Для этого продумываем интерфейс функций:

- какие параметры передавать?
- нужно ли возвращаемое значение?

Динамические массивы: вспомогательные функции

Если в процессе написания кода программы возникает необходимость создания больше одного динамического массива, неплохим решением будет вынос подзадач по выделению/высвобождению памяти в отдельные функции (та самая *декомпозиция*).

Для этого продумываем интерфейс функций:

- какие параметры передавать?
- нужно ли возвращаемое значение?

```
1 int* allocate(size_t count);
2 // vs
3 void allocate(int *& ptr, size_t count);
```

Динамические массивы: вспомогательные функции

Далее реализуем функции для выделения/высвобождения памяти для одномерных, двумерных и трёхмерных массивов типа **int**

```
1  void allocate(int *& ptr, size_t count);
2  void deallocate(int *& ptr);
3
4  void allocate(int **& ptr, size_t rows,
5                size_t cols);
6  void deallocate(int **& ptr, size_t rows);
7
8  void allocate(int ***& ptr, size_t rows,
9                size_t cols, size_t layers);
10 void deallocate(int ***& ptr, size_t rows,
11                  size_t cols);
```

Динамические массивы: вспомогательные функции

Одномерный случай

```
1  void allocate(int *& ptr, size_t count)
2  {
3    ptr = new int[count];
4  }
5
6  void deallocate(int *& ptr)
7  {
8    delete[] ptr;
9    ptr = nullptr;
10 }
```

Динамические массивы: вспомогательные функции

Двумерный случай

```
1  void allocate(int **& ptr, size_t rows,
2                size_t cols)
3  {
4    ptr = new int*[rows];
5
6    for (size_t i = 0; i < rows; i++) {
7      ptr[i] = new int[cols];
8    }
9  }
10
11 void deallocate(int **& ptr, size_t rows)
12 { if (ptr == nullptr) { return; }
13   for (size_t i = 0; i < rows; i++) {
14     delete[] ptr[i];
15   }
16   delete[] ptr;
17   ptr = nullptr;
18 }
```

Динамические массивы: вспомогательные функции

Трёхмерный случай

```
1  void allocate(int ***& ptr, size_t rows,
2                size_t cols, size_t layers)
3  { ptr = new int**[rows];
4    for (size_t i = 0; i < rows; i++) {
5      ptr[i] = new int*[cols];
6      for (size_t j = 0; j < cols; j++) {
7        ptr[i][j] = new int[layers];
8      }
9    }
10 }
11
12 void deallocate(int ***& ptr, size_t rows,
13                 size_t cols)
14 { if (ptr == nullptr) { return; }
15   for (size_t i = 0; i < rows; i++) {
16     for (size_t j = 0; j < cols; j++) {
17       delete[] ptr[i][j];
18     }
19     delete[] ptr[i];
20   }
21   delete[] ptr;
22   ptr = nullptr;
23 }
```

Динамические массивы: вспомогательные функции

И пример применения функций:

```
1  {
2    const size_t rows = 4, cols = 5;
3    int** my_matrix = nullptr;
4    allocate(my_matrix, rows, cols);
5
6    printf("Matrix %zu x %zu\n", rows, cols);
7    for (size_t i = 0; i < rows; ++i) {
8      for (size_t j = 0; j < cols; ++j) {
9        my_matrix[i][j] = pow(0.5, i + j);
10       printf("%d ", my_matrix[i][j]);
11     }
12     printf("\n");
13   }
14   printf("\n");
15
16   deallocate(my_matrix, rows);
17 }
```

Утечка памяти: как появляется?

```
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], rate;
8    // Сложная обработка массива
9    // и нет вызова оператора delete
10
11   return rate;
12 }
13
14 get_complex_rate(5000000);
15 get_complex_rate(222000);
```

Утечка памяти: исправление и внимательность

```
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], rate;
8    // Сложная обработка массива
9    delete[] arr;
10
11   return rate;
12 }
13
14 get_complex_rate(90000);
15 get_complex_rate(1500000);
```

Динамические массивы: вспомогательные функции

Пример - изменение размера одномерного массива

```
1  void reallocate(int *&ptr, size_t sz, size_t new_sz)
2  {
3    if (ptr == nullptr) { return; }
4    if (sz == new_sz) { return; }
5
6    size_t least_sz = (sz < new_sz) ? sz : new_sz;
7
8    int *new_arr = new int[new_sz];
9    for (size_t i = 0; i < least_sz; i++) {
10     new_arr[i] = ptr[i];
11   }
12   delete[] ptr;
13   ptr = new_arr;
14 }
15
16 int *my_arr = new int[40];
17 //Расширить динамический массив my_arr до 65 элементов
18 reallocate(my_arr, 40, 65);
```

Указатель на функцию (function pointer)

Указатель на функцию

Указатель на функцию - указатели специального типа, позволяющие использовать функции языка как переменные. Их основные характеристики:

- позволяют передавать функции как аргументы в другие функции (без возможности быть параметром со значением по умолчанию);
- позволяют объявлять массивы функций, одинаковых по типу возвращаемого значения и со совпадающим списком аргументов;
- позволяют делать отложенный вызов функций;
- не требуют разыменования;
- не требуют явного присвоения адреса существующей функции.

Общий синтаксис:

```
<тип_возвращаемого_значения>
            (*<имя_указателя>) (<типы_аргументов>);
```

Указатель на функцию

Пример:

```
1  // Создаём указатель на функцию, которая
2  // возвращает значение типа double и
3  // имеет один параметр типа double
4  double (*func_ptr)(double) = nullptr;
5  //....
6  double arg = 3.14 / 2;
7  func_ptr = sin; // указываем на sin из <cmath>
8  printf("sin(%.4f) = %.7f\n", arg, func_ptr(arg));
9
10 func_ptr = log; // указываем на log из <cmath>
11 printf("log(%.4f) = %.7f\n", arg, func_ptr(arg));
```

Указатель на функцию

Уже было: передача функции сравнения в функцию сортировки

```
1  bool my_compare(int left, int right)
2  {
3
4    return abs(left) > abs(right);
5  }
6
7  int arr1[] = { 3, 1, 5, 4, 3, 2,
8                 1, 8, 4, 76, 4, 67 };
9  std::sort(arr1, arr1 + 12, my_compare);
10
11 printf("After sorting: ");
12 for (int elem : arr1) {
13   printf("%d ", elem);
14 }
15 printf("\n");
```

Указатель на функцию

Прокачаем одномерный allocate

```
1  void allocate_with_init(int *& ptr, size_t count,
2       void (*init)(int *, size_t))
3  {
4    ptr = new int[count];
5
6    if (init != nullptr) { init(ptr, count) }
7  }
8
9  void seq_from_one(int *ptr, size_t count)
10 {
11   for (size_t i = 0; i < count; i++) {
12     ptr[i] = i + 1;
13   }
14 }
15
16 int *arr_1d = nullptr, *other_1d = nullptr;
17 allocate_with_init(arr_1d, 10, seq_from_one);
18 allocate_with_init(other_1d, 10, nullptr);
```